ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Systems@ETH Zürich

Bachelor's Thesis Nr. 158b

Systems Group, Department of Computer Science, ETH Zurich

Barrelfish NIC driver in Rust

by
Jakob Meier

Supervised by
Timothy Roscoe, Gerd Zellweger, Roni Haecki

September 2016 - March 2017

inf | Informatik
Computer Science

Abstract

By implementing a network interface controller (NIC) driver on top of the research operating system (OS) Barrelfish, the new systems programming language Rust and the implications it brings for development have been analyzed.
One feature that makes Rust attractive is compile-time memory safety in concurrent environments. Most importantly, it is achieved without garbage collection, and there is no need for explicit memory allocation, either.
Performance comparisons to an equivalent driver written in C show that Rust does not come with an overhead as many other languages with safety features do.

Contents

1 Introduction

Motivation: Today's technology is dominated by multi-core machines and with the Internet of Things (IoT), devices are becoming more complex in general. Even embedded systems are supposed to deal with network protocols. Although this trend is not really new, it is not old either when comparing it to the lifespan of the dominant programming languages used for programming virtually any sort of device.
The referred languages are C, dating back to 1972, and C++ introduced in 1983. Both of them are famous for being "hacky". They allow programmers to modify the hardware in a direct way. They are also known for causing bugs deep down at the base of a system.
While the most recent standard for C is from 2011, a new standard for C++ should be released in 2017, and further standards are planned already.[28]
A question addressed in this thesis is why C and C++ are still that dominant. Have tools for development not been improved in the meantime, and would it not be advantageous to migrate to more modern languages?

Scope: To analyze that question, the focus of this thesis is on a new programming language named Rust that could potentially be an alternative.
In section 2.3, I give a basic introduction to the language itself.
In theory, Rust offers everything that C does, and much more. Compile-time memory safety, even among multiple cores, without a garbage collector, is one of the interesting features the new language brings with it. Another nice feature is the functional aspect integrated into Rust with closures and pattern matching. But also some features that remind of object oriented programming (OOP) found their way into Rust.
In practice, however, there could be some migration problems when developing with Rust. Especially when there is an existing code base completely written in another language. In addition, programmers are concerned about productivity, performance, and safety. Since Rust is fundamentally very different from C and C++, it is unclear how these key attributes are affected by using Rust over the other languages.

Project: To explore named issue, I have implemented a NIC driver completely in Rust and integrated it into the research OS Barrelfish which itself uses C for all low-level code. A brief introduction to Barrelfish is given in section 2.1
While using Rust in a very practical way, I could see different problems arising with the young language, but I did also have many positive experiences with it. Those which I found interesting enough to mention are documented in sections 3 and 4 of this thesis.
The finished driver is then compared to a driver that has been written in C, also for Barrelfish and for the same hardware. How the performance compares between the two implementations is discussed in section 4.5.3.

2 Background

2.1 Barrelfish

Barrelfish[5][1] is a research OS, built from scratch by the Systems Group at ETH Zurich. With its characterizing multikernel approach, it scales much better than traditional OSs architectures when it comes to high numbers of cores. Having only one single kernel managing resources for the entire system makes it a natural a bottleneck that is hard to avoid as programmer.
In Barrelfish, there is more than one kernel. In fact, Barrelfish runs one independent kernel on each core. To connect them to a complete system, a lot of logic goes into user space services and communication between cores happens through messages managed by those services, as opposed to shared memory. With that design, there is virtually no limit on the number of cores within the system that can work together logically.
Barrelfish runs on different platforms and the development team is porting it to new platforms as technology evolves. That, together with the natural independence of the kernels, makes it possible to unite heterogeneous cores in one single OS. In other words, processors from different architectures can be combined within one system running a single instance of Barrelfish.
Barrelfish is in an ongoing development and serves as a research platform for different projects. Two examples, both involving networking, are mentioned in sections 6.1.3 and 6.1.2.
The remainder of this section will provide a quick overview on those parts of the OS which have been relevant for me to build a network card driver.

2.1.1 Kernel- and User-Level

In Barrelfish, the OS code running on each core is divided into two parts. First, the *CPU-driver*. It contains everything that needs to run in privileged-mode, and is highly dependent on the underlying architecture. Each CPU-driver works solely locally, i.e. it does not share any resources nor does it communicate with other cores directly. In essence, the terms CPU-driver and kernel are synonyms in Barrelfish.
Second, there are *Monitors*, which run completely in user-mode. This is where all the independent cores are combined to one distributed OS. Agreement protocols are used to replicate the state of the system and thereby providing the usual mechanisms and policies of an OS. For example allocating memory is done in a Monitor and in order to do that successfully, the Monitor has to ensure a system wide consistency among all replicas of the allocation table.

2.1.2 Resource Management: Capabilities

Allocation and management of memory resources is all done in user-level space, even for the kernel objects. This is one of the properties which makes Barrelfish distinct from traditional OSs and allows the distributed multikernel approach. There are certainly different ways of achieving this in a sound way, but what has been implemented for Barrelfish is a system of so called *Capabilities*.[26]

[1] http://www.barrelfish.org/

All memory management actions in Barrelfish happen through system calls, which manipulate Capabilities. These are user-level references to kernel objects or regions of physical memory. In that way, all the work, like allocating and manipulating page tables, is still done in user-level and the CPU driver will essentially only check the correctness of the operations. This concept of Capabilities is not new, the model has been taken from seL4[17]. However, to use it in distributed multi core environment, such as Barrelfish, some generalizations were necessary. To name one modification, in Barrelfish some Capabilities can also be sent across cores which has not been of any concern in the single-core environment of the seL4.

2.1.3 Building: Hake

The build system called *Make*[2] is one of the most common tools to handle complex projects where many files need to be compiled together. Instead of manually invoking the compiler(s) with all the different input files and parameters, Make-rules can be used to do this automatically. Make also keeps track of which files of the projects have been changed and which have not, making recompilations much faster.
In a large system like an OS, the number of required Make-rules is growing rapidly as development continues. Even just adding a set of Make-rules for each new file becomes tedious quickly. To resolve that issue, the Make-rules themselves can be generated by yet another build system. For that purpose *Hake* has been created for Barrelfish. [22]
Hake is written entirely in Haskell and its use is relatively straight forward. A configuration file for each program sets up all parameters. Hake will then automatically find the configuration file and generate the Make-rules.

2.1.4 Device management: Kaluga

Generally speaking, the list of connected devices to the machine on which an OS will be running on, is not known at compile time. Therefore, a discovery of devices and a mapping to corresponding device drivers is a task an OS is supposed to do.
In Barrelfish this job is accomplished by the combination of a system knowledge base (SKB) and a device manager called *Kaluga*[29].
To implant a driver to Barrelfish, an entry in the SKB with the information of the type of devices it can handle is required. Once this is done, Kaluga will automatically run the driver when a fitting device is connected to the system.

2.1.5 The network stack

In Barrelfish, networking is done entirely in user-level space. It is divided into different user space libraries which combined provide all functionalities expected from an OS.
The core of the network stack, where all the standard protocols like TCP, IP, UDP, DHCP, DNS, ARP and ICMP are implemented, is an implementation of **lightweight IP (lwIP)**[8] for Barrelfish. As the name suggests, the focus of lwIP is to use as few resources as possible to support the above list of basic networking protocols. It is therefore even suited for embedded systems.[13]
On top of lwIP we have the **networking daemon (netd)**, an important user-level library and also part of Barrelfish. A single instance of netd is started at boot time and makes sure networking actions are performed constantly. For instance, it triggers DHCP and manages

[2] https://www.gnu.org/software/make/

ARP caches. It can also be configured with network configurations such as a static IP address, a gateway, and a network mask.
To connect a NIC to the OS' part of the network stack, another user-level library of Barrelfish called **NetQueueManager (NQM)** comes into play. It provides an interface to register hardware queues of the NIC by calling a function called *ethersrv_init()* which takes a bunch of callback function pointers as arguments. Using these callbacks, the NQM will then feed the driver who registered them with physically mapped memory buffers for direct memory access (DMA). Both sending and receiving is implemented in that way. The NQM can also terminate queues when they are no longer used.

2.1.6 Describing devices: Mackerel

Mackerel[23] is a domain specific language (DSL) that is used in Barrelfish to describe registers and address spaces of hardware devices. With a simple and intuitive syntax, it allows to describe complex semantics on a bit field level for registers. Following this description, register read and write functions are generated, which respect all defined constraints. Hence, in application code, all necessary bit-shifting and bit-masking is hidden behind a call to a generated Mackerel function.
The compiler which converts Mackerel files to C code functions is written in Haskell. As part of porting Rust to Barrelfish[9], a back-end generating Rust code has been added. The possible inputs remained the same Mackerel files, but the output can now be either C or Rust code.

2.2 The BCM5704C Gigabit Ethernet card

The driver developed during this project operates on the Broadcom BCM5704C Gigabit Ethernet chip which has been designed for high-density server applications. At Eidgenoessische Technische Hochschule Zuerich (ETHZ) we have about a dozen rack machines with that card integrated. (See also the hardware specification in the appendix, section 9.3.)
The following block diagram shows the different components of the card. The diagram has been taken from the official documentation, the programmer's guide[6].

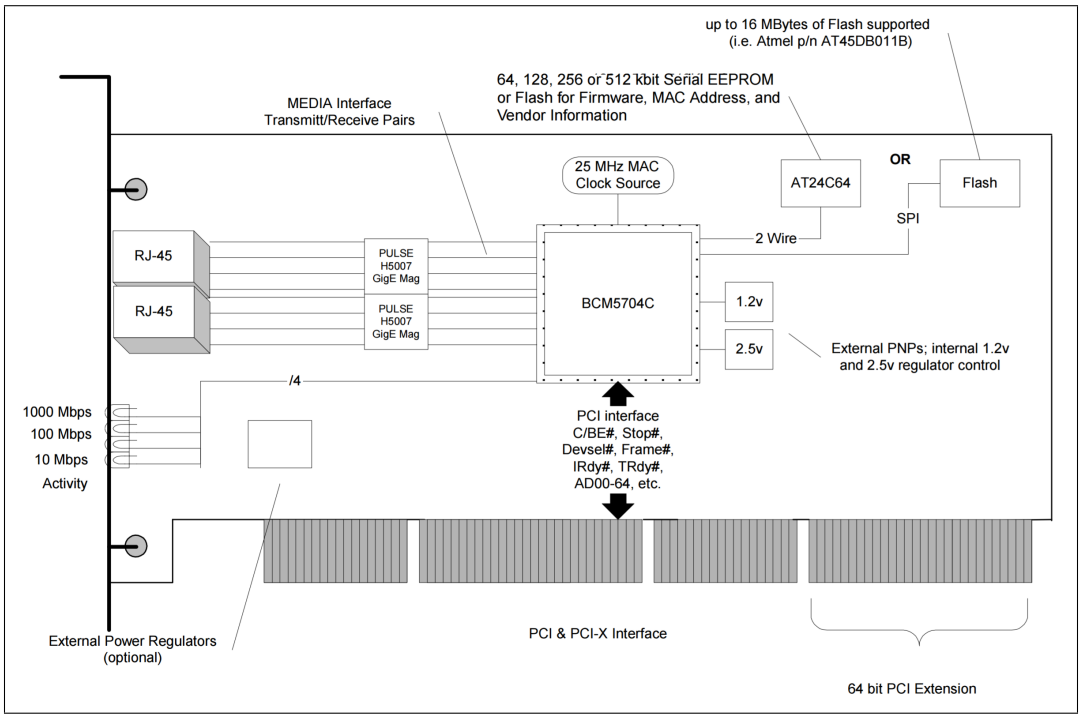

Figure 1: The BCM5704C board block diagram taken from figure 7 in the programmer's guide

2.2.1 Features

The Broadcom 5704C combines two triple-speed Media Access Controls (MACs) with integrated transceivers in one chip which can either work independently, with one of the two turned off, or teamed as one. Triple speed in this context means, being able to handle three different transmission rates, 10, 100, or 1000 Mbit/s on each of the two LAN ports.
Each MAC has its own Reduced Instruction Set Computer (RISC) processor and 64 KB on chip buffer memory.
The chip offers many optional features. The most notable among them include jumbo frames, VLAN tags, layer 2 priority encoding, programmable receive rule checker, Wake on LAN (WOL) support and TCP/UDP/IP checksum offloading. [6]

2.2.2 Existing drivers

There are drivers supporting the entire Broadcom NetXtreme BCM57xx Gigabit Ethernet family for Windows, Linux[3] and FreeBSD[4]. Both support the greater part of features mentioned in section 2.2.1, but there is no driver implementing all of the features. For instance, the card can use up to 16 receive rings and up to 16 transmit rings, yet all existing implementation that I know of use them combined as one single ring.[10]
Two years before my own bachelor thesis, Philipp Gamper implemented a first driver running on Barrelfish for the exact same card. [10] Thanks to his effort, I had a reference implementation in C code that shows how the general driver design could look in the Barrelfish system.

2.2.3 Data flow

The general theme for data flow in NICs is always similar. This section is a summary of that and points out implementation details of the BCM570C .
When the OS has data to send, it will be prepared as Ethernet frames and stored in buffers of a size the MAC can handle. A descriptor holding all relevant information about the buffer will be given to the driver instance. This descriptor is then enqueued in a *buffer descriptor ring*. This data structure is shared with the MAC, for the BCM57xx family in particular it is replicated through DMA. Once the MAC notices that a buffer is available, it will take the prepared frames and pass it on to the integrated PHY transceiver. The access to the buffer happens through DMA, thus the buffers need to be physically mapped.
After sending has completed, the buffer descriptor ring is updated, which releases the described buffer to be used again for new data.
Although receiving works similar, it requires two buffer descriptor rings instead of just one. The reason is that frames arrive in an unpredictable way and if the MAC has no memory available to write, the data is dropped after the internal buffer is full. Thus we cannot allocate buffers upon arrival of the data, instead we should allocate the data with foresight.
A *receive ring*[5] is used to queue up buffer descriptors to allocated host memory. Again using DMA, the MAC can write the incoming data immediately into those buffers when they arrive.
After filling a buffer, the descriptor is removed from the receiver ring and inserted into the other buffer descriptor ring, the *return ring*[6].
The driver waits for changes in the return ring caused by the chip and delivers the data to the OS.
Note that the receive ring is located in host memory and also replicated from the driver to the chip using DMA. The return ring is stored in host memory, too, but no replication is required. The chip will just cache the producer index and update the ring directly in host memory using DMA.
As mentioned in section 2.2.1, the card also allows to make IP and UDP/TCP checksum calculations on the card, to reduce CPU overhead. Specifically, it can optionally check the correctness of arriving packets and compute the checksum for the transmitted pack-

[3]Tigon III: https://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/tree/drivers/net/ethernet/broadcom/tg3.c?id=refs/tags/v4.0.2
[4]Broadcom BCM57xx/BCM590x Gigabit/Fast Ethernet driver: https://www.freebsd.org/cgi/man.cgi?query=bge&sektion=4
[5]Also called Receive producer ring
[6]Also called Receive return ring

ets, thereby interfering with the network and transport layers. On the data link layer, the frames' checksums have to be checked and computed by the MAC either way.

2.2.4 Data structures

Next, let's have a closer look at some of the data structures defined by the BCM5704C to allow the data flow discussed in section 2.2.3. We will go through three types of structures bottom up.

Buffer Descriptor

The actual data exchange with the OS happens exclusively over buffer descriptors. The driver itself never looks at the data inside of the buffers, its sole reason of existence is to make sure the NIC knows where those buffers are and to keep trak of the state of each buffer.
The OS has its own representation of buffers, in which format the driver receives control over them. Next it will create a corresponding buffer descriptor, which is a representation of a buffer that the NIC understands. The descriptor contains just enough information to find and use the buffer and is very small compared to the buffer size. Thus, moving around the descriptor is much faster than moving the buffer itself.

Offset (bytes)	*31*	*16 15*	*0*
0x00	Host Address		
0x04			
0x08	Length	Flags	
0x0c	Reserved	VLAN Tag	

Figure 2: Send descriptor format taken from table 29 in the programmer's guide

Offset (bytes)	*31*	*16*	*15*	*0*
0x00	Host Address			
0x04				
0x08	Index		Length	
0x0c	Type		Flags	
0x10	IP_Cksum		TCP_UDP_Cksum	
0x14	Error_Flags		VLAN tag	
0x18	Reserved			
0x1C	Opaque			

Figure 3: Receive descriptor format taken from table 35 in the programmer's guide

The BCM57xx family defines two types of descriptors, one for sending one for receiving. Their exact structures are shown in figures 2 and 3. Basically, they contain the physical address of the buffer, the length of the stored data, some flags for configuration and other values for optional features like VLAN tags or checksum calculations.
There is also an opaque field for the receiver. The idea behind this 32-bit value is to store

the identifier for the buffer given by the OS. The value will be preserved when the descriptor goes from the receive ring to the return ring, thus when the driver is looking at a buffer descriptor in the return ring and delivers it to OS, it can also tell the OS the identifier of the buffer.
However, the identifier used by Barrelfish and the NQM is 64 bit wide. Because that does not fit into the opaque field, it is not used in my implementation.

Buffer Descriptor Rings

To manage the buffer descriptors, they are stored in producer-consumer rings. Each ring stores descriptors in a underlying fixed size array. Two indices, the producer and the consumer index, determine which slots of the array are currently in use. The producer rings always point to the first empty slot, the consumer index points to the first valid descriptor ready to be consumed. When the two indices point to the same location, then the ring is empty, i.e. all descriptors are invalid. When a new descriptor is inserted, the producer ring is increased with a possible wrap around to stay inside the array boundary. When the producer index reaches the slot one behind the consumer index, the ring is considered to be full.
How the three different types of ring work together is described in section 2.2.3. In theory, there are three different types of receive rings, one each for mini, normal and jumbo packets. Since my implementation only supports normal sized packets, I will pretend there is only one type of receive ring.
The rings themselves as defined by the BCM57xx family are basically only arrays of send or receive buffer descriptors and the indices are stored independently. However, there is a structure called *Ring Control Block (RCB)* for each ring that holds some meta data. The fields of an RCB are shown in figure 4.

Offset (bytes)	*31*	*16*	*15*	*0*
0x00	Host Ring Address			
0x04				
0x08	Max_len		Flags	
0x0c	NIC Ring Address (reserved in BCM5705)			

Figure 4: RCB format taken from table 26 in the programmer's guide

The first field is a physical address pointing to the host memory location where the buffer descriptor array is stored. The flags are for configuration purposes.
The max_len field has different semantics in different contexts. In our case, it names the size of the descriptors array for send and return rings. For receive producer rings, it is the size of the buffers described by the buffer descriptors.
The meaning of the last field is again dependent on the type of ring. It is only valid in receive producer rings and in send rings, not in return rings. It points to the internal memory location for the buffer descriptor cache.
All RCBs must be written to the corresponding location in the NIC's internal memory. For the receive ring that is a 16-byte register located in the register block. For send and return rings, the RCBs are located at different position in the local memory block. A lot of care is required from programmers here, it is vital to represent the RCBs exactly as defined in the

specification and store them at the right address. The chip relies on the values written in there to find and use the rings allocated in host memory.

Status Block

The last data structure I want to talk about is the status block. It is used to coordinate the indices of the descriptor rings between the host and the chip. The status block resides in host memory and is continuously updated through DMA by the NIC. It contains the following fields:

- A 32-bit status word. Only 3 bits are used. They signal link state changes, updates to the status block and errors, where the error bit is an OR of a long list of other error flags.
- A 32 bit status tag, increased with each status block update. Only 8 bits are used. This can be used for refined interrupt handling.
- The receive ring's consumer index.
- The send ring's consumer index.
- The return ring's producer index.
- Some reserved fields.
- Indices of other rings, if implemented.

When reading the status block, the host is expected to clear the *updated* bit in the status word.
Note that the indices stored in the status block are exactly those which the NIC updates to inform the driver about its state. For instance when a packet has been received, the receive ring's consumer index will first go up and next, as soon as the full packet is ready, the return ring's producer index will be increased as well. The driver can observe the progress by reading the changes in the status block.

2.2.5 Memory regions

The BCM5704C has two important memory regions, called *Register Block* and *Memory Block*. Both internal RISC processors have their own view to those regions as one large contiguous memory range, independent of the host's view. Figure 5 shows this internal view. But the two blocks are not necessarily tied together. The host can decide between different views on those regions and access them in different ways. Understanding the different views and involved address mappings is crucial to implement a driver. The relevant modes are discussed in section 2.2.6 following right here.

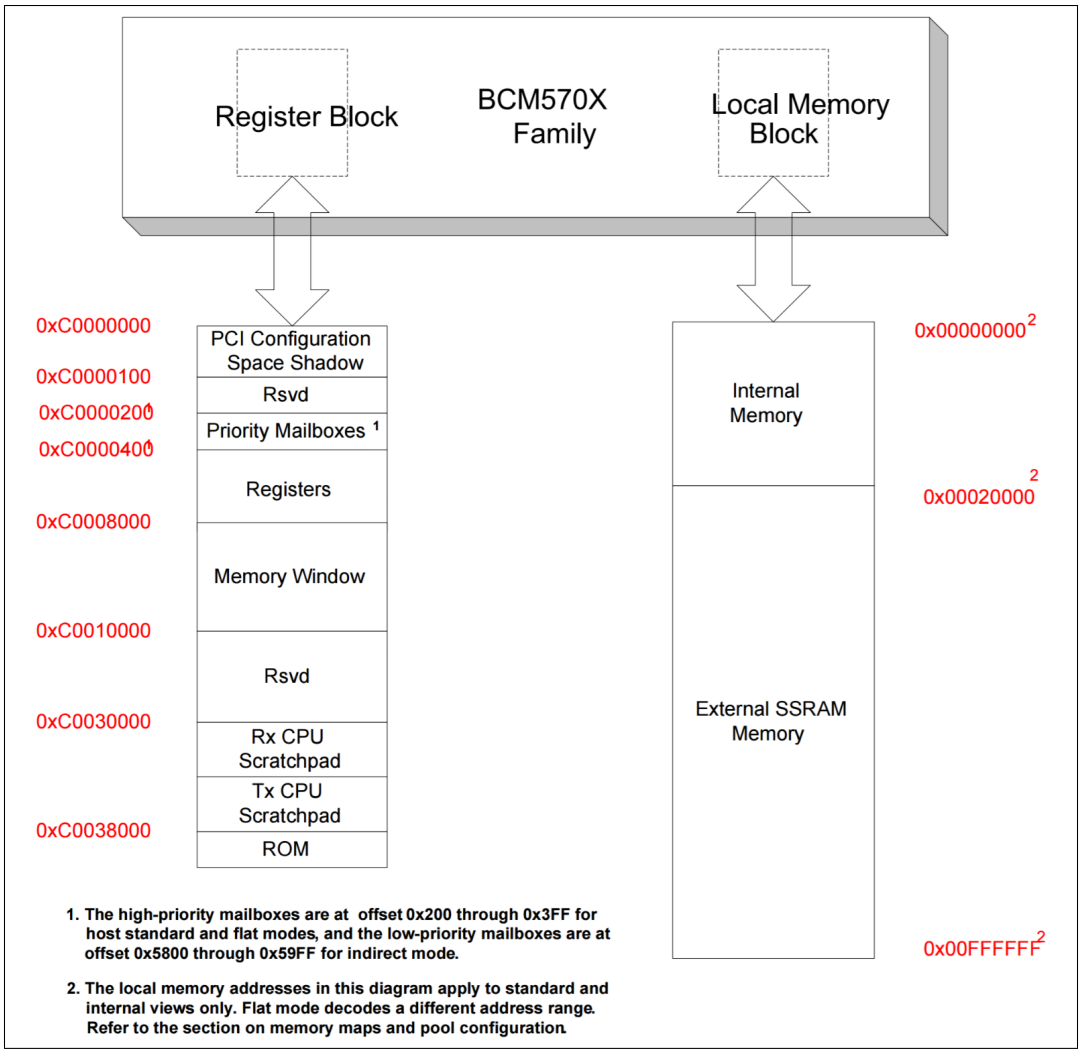

Figure 5: Local contexts from figure 63 in the programmer's guide

2.2.6 Communicating with the chip

Peripheral Component Interconnect (PCI)

To discover and connect the device to the system, PCI is used. PCI devices have to provide a number of required header registers in their PCI configuration space. Information about the vendor, the device, and the revision ID can be found there. Of special interest for device driver developers, the Base Adress Register (BAR) is also located there.
The rest of the PCI configuration space is device specific and used by the BCM57xx family to configure and change the operational state of the MAC. Beside various control registers which allow to make those configurations, there are some more interesting registers located in this region which are used for different types of communication with the card.
For a full schema of the PCI configuration registers, see figures 11 and 12 in the appendix.

Universal Network Device Interface (UNDI) Mailbox Access

There are three mailbox[7] registers in the device specific part of the PCI configurations space which directly shadow[8] registers in the register block. They hold the three indices that host software needs to update when handling receive and transmit data flow. Namely, they are the producer index of the standard receiver ring, the consumer index of the return ring and the producer index of the send ring. Assuming we only use one ring of each type, those registers are enough to inform the MAC when the host has consumed receive buffers and when it has available buffers for sending or receiving.
These shadow mailbox registers can be read and written, and they are typically used by UNDI drivers to provide basic network connectivity without the need of memory mapped registers.

Indirect mode

The indirect mode is one way to access registers and local memory of the MAC. It is completely independent from other modes and can be enabled using a bitfield in a control register in the PCI configuration space.
Using the indirect mode works with two pairs of Address-Data registers, which are also located in the PCI configuration space. One pair is for accessing the register block, the other one for the local memory block. In both cases, the address register must be set to a valid position first, then the data register can be read or written to access the memory pointed to by the address currently stored in the address register.
It is worth noting that when actually implementing this memory accesses, there are certain restrictions on the range of the address register, depending on which block is accessed and the specific hardware configuration. Also, programmers must be careful when combining this mode with other modes. For instance, an additional read must be inserted between two consecutive writes in indirect mode when the register pair is accessed through memory mapped mode instead of a direct PCI configuration space access.

Memory mapped mode

This is also referred as direct mode, as opposed to the indirect mode discussed in the previous paragraph. The BCM57xx family allows for two different memory mapped modes, called flat and standard mode. The former is mostly useful for diagnostic tools and requires a large amount of memory address space. I will only discuss the standard mode here.
In standard mode, 64KB of host memory space is mapped to internal resources of the MAC. More specifically, the MAC asks for physical addresses spanning a range of 64KB that are not in use. The PnP BIOS or OS will then write the base address for this region to the BAR mentioned in the paragraph about PCI.
After that, host software can access the physical address region to manipulate the register block. Accessing the local memory block is also possible, in this case using the *memory window*. Similar to indirect access, there is a register in PCI configuration space that acts as cursor, pointing to a memory region. But instead of a data register, there is an entire 32KB large memory window shadowing the data in the local memory block pointed to by the cursor. The cursor uses the address space internal to the MAC and must always align in 32K boundaries. Its range is limited by the memory available to the hardware. For instance, 0x0 and 0x8000 are valid addresses for any hardware configuration, whereas 0x100 is invalid

[7] A mailbox register is one that will trigger some event inside the chip when written.
[8] A *shadows* B in this context simply means that accesses to region A is redirected to region B.

because it is not aligned properly.
Figure 5 shows the internal view of the local memory block and helps understanding which locations are valid for the cursor. In the same figure it is visible that the PCI configuration space is also shadowed with the memory mapping of the standard mode. This allows to read and write those registers using the slightly more efficient memory mapped access instead of the normal PCI configurations space access.

PHY registers

The integrated transceivers of the BCM57xx family are handling the physical layer of the OSI model, and they are accessible indirectly through the MACs on a BCM5704C. Here is only explained how it works with this specific chip, since the exact process differs among chips of the same family.
Each transceiver has its own, tiny address space, containing 32 registers of size 16-bit. To access these registers, programmers can use the Management Interface (MI) communication registers of the MAC. The process involves more than one step and the access to the MI communication register itself is possible in indirect mode or memory mapped mode.
In a first step, MI auto-polling must be turned off. Then a 32-bit value is written to the MI communication register which contains four encoded values:

1. Which of the two PHYs should be accesses.
2. The address of the register in the transceivers address space.
3. Whether it is a read or write command. Only one is possible at once, combining them is undefined behavior.
4. In order to write to a register, the first 16 bits should contain the value to be written.

After that, the driver waits until a specific bit is cleared, indicating that the transaction has completed. If it was a read-command, the first 16 bits of the MI communication register will then contain the content of the register. Finally, we may have to enable MI auto-polling again.
If accessing PHY registers as described above fails, it will be marked in a bit flag of the MI control register.

2.2.7 Interrupts

The BCM5704C can be configured to emit either legacy interrupts or follow the Message Signaled Interrupt (MSI) mechanism going through the PCI bridge. For this project, the former has been used.
When an interrupt is generated by the NIC, the OS will call the interrupt handler registered by the driver. The interrupt is then acknowledged by the interrupt handler through a mailbox register. The card will know from that, that the host software is currently handling an interrupt, and it will therefore not generate more interrupts. Also, different general settings for the card are configurable with two values, one for the normal case and one that applies whenever an interrupt is being handled. One such example is the frequency with which the DMA engine replicates data between the host and the card.
After the interrupt has been acknowledged, the driver will check the reason for the interrupt. Usually new data has arrived, but it could also be that the error flag has been set for some reason. By looking at the status block, the drive can find that out and simultaneously look

up the updated index for the ring structures.
The actual job is then to provide the arrived data to the waiting user process and updating involved indices of the ring structures.
Next, the driver would like to simply enable interrupts again and wait for the next interrupt. But there is a chance that the driver has missed an event that usually generates an interrupt, which has been suppressed because the driver was busy handling the previous interrupt. For that reason, the driver checks the status block again after re-enabling interrupts. If it has been updated, interrupts are disabled again immediately and the new event is handled.
With legacy interrupts, additional care is required because interrupts do not go through the PCI bridge. That means, it is possible that the interrupts arrives and the data on the PCI bridge has not been flushed yet, therefore the driver might read an old value of the status block. To avoid that, a dummy read of some data inside the card is inserted which forces the PCI bridge to be flushed and thereby assures the status block in host memory is up to date.
With MSI, that problem does not exists since the interrupt itself goes through the PCI bridge and enforces the flush.

2.3 Rust

2.3.1 Overview

Background: Rust[9] is a programming language created at Mozilla Research and is today developed by an open community which has reached a size of more than a thousand contributors. The first stable release was published in May 2015, but the development has not come to a stop, yet.

Multi-paradigm: The language Rust combines a variety of paradigms, it is statically typed, uses type inference and has a form of polymorphism. It is not an object oriented programming language, but it enables almost the same abstractions as known from OOP. Further, it comes with features like closures and pattern matching which are typical for functional languages.

Safety features: One of Rust's key features is that it provides strong compile-time guarantees. Memory and thread safety are default properties of programs written in Rust, although the language provides the possibility to mark code as unsafe which will allow that code to violate some of those properties. Especially for systems programming, this possibility to escape the strong boundaries of the language's features is crucial. Nevertheless, the safety features are very useful even in systems programming, as they give programmers the possibility of reasoning about what parts of their code base can yield which kind of errors.

Easy memory management: The memory model introduced in Rust allows programmers to omit memory deallocation, just like in the presence of a garbage collector, except that there is no garbage collector in Rust.[10] Instead, the Rust compiler figures out at compile-time at which point it is safe to deallocate the memory.

Zero-cost abstractions: Even though the language provides high-level abstractions and strong safety guarantees, it aims for a performance viable for systems programming. Runtime costs caused by the type system are avoided with the approach of handling as much as possible at compile-time.

Research: Rust got some attention from different areas in computer science, particularly in systems programming it has been tested and studied by a number of people. I will present some of the results in section 6.3.

Rest of the chapter: To understand the code samples used in this thesis, a basic understanding of the programming language Rust is required. The rest of this chapter aims to give just enough information about the language that all examples can be understood by a reader who never saw Rust code before. However, it is by far not a complete introduction to the language. To learn to program in Rust, I can warmly recommend the official online Rust book [11] or for a different, more practical approach, the Rust by Example website.[12]

[9] https://www.rust-lang.org/

[10] Early versions of Rust did mention a possible optional garbage collector for the future, but they abandoned that idea completely in 2014 and there has never been an actual garbage collector in Rust.[1]

[11] https://doc.rust-lang.org/stable/book/

[12] http://rustbyexample.com/RustByExample

2.3.2 Basic syntax

Primitive types and casting

The most simple types, integers and floats, are represented in Rust as a combination of a letter and the bit width. For instance, **i32** stands for a signed 32-bit integer, **u8** is a 8-bit unsigned integer and **f64** is a floating point number using 64 bits. Furthermore, there are numeric types **isize** and **usize** which are integer whose bit-width depends on the underlying architecture, equivalent to *size_t* known from C99. Lastly, there are the primitive types **bool** and **char**[13].

To cast between types, the keyword **as** is used. An example of this is shown in the second line from behind of code block 2, where a **u32** and a **i32** are both casted to a **f64** before division.

Not covered so far are strings. There are different representations for strings in Rust and I will not explain them here. How strings are used in the code should be easy to understand, even without knowledge of all the different representations and their use cases of strings in Rust.

Variable bindings

Variables are defined using the keyword **let** and subsequent **let** statements with the same variable name can overwrite, or shadow, old bindings. (see code block 1)

In Rust, there is a difference between mutable and immutable bindings. That is, if a variable is not defined as mutable, then its value can only be read, modifications are not allowed. Note that an initialization on a immutable variable is allowed, but afterwards the value cannot be changed again. For instance in Java, we have the same semantic with **final** variables, but in Rust it is the default for any variable. If we want to modify it, we have to define the variable with the **mut** keyword.

Rust uses type interference by default, comparable to the **auto** keyword in C++11. In other words, the type of a variable does not have to be specified explicitly. In most cases the compiler can derive the correct type by itself. Never the less, Rust is strongly statically typed.[14] This can lead to some confusion for programmers new to the language, as there are compiler errors about mismatched types which have not been declared explicitly anywhere. (see the last line of code block 1)

Functions and macros

Function in Rust are defined with the keyword **fn** and the parameter list is given in the form *(name1: type1, name2: type2, ...)*. A return value can optionally be defined with an arrow: *->return-type*

Although Rust has a **return** statement, it is no necessary to use it when returning a value from a function. In Rust, when an expression is not terminated with a semicolon, it will pass the value to the surrounding expression. In particular, when the last line of a function is not terminated with a semicolon, its value will act as return value. (see code block 2)

In code 2 we can also spot the printing mechanism of Rust: *println!(...)* It might look like a ordinary function call, but it is in fact a macro call. The *println!(...)* macro works basically just like a *println(...)* function in C. The first argument is a string and any additional argument has to be a value that can be converted to a string. Inside the first string we

[13] One **char** in Rust is 4 bytes, because it uses Unicode encoding.

[14] In contrast to dynamically typed languages like Python where the types are only checked at runtime.

can put some place holders[15] for the arguments and there are some options to modify the representation.
Macros in Rust are marked with the explanation mark after the name and they allow for more general argument lists. A normal function in Rust has a specific set of types for its parameters, which must be met by all invocations[16]. A macro on the other hand, can be defined for different types and the number of parameters can be dynamic. Beside the explanation mark, macros are called exactly like functions and this will be enough to know about macros for us. But it is worth mentioning that Rust's macros are much more than just a way to have dynamic parameters.
Another macro will have its appearance in section 2.3.8 about inline assembly code. For more information about macros in general I can once again recommend the Rust book.

```
fn main(){
    let a = 5;
        // a = 4;
        // Compiler error: re-assignment of immutable variable `a`
    let a = 4; // No error
    let a = "Hello world"; // No error

    let mut b = 0; // A mutable variable binding
    b = 7;
    // b = 7.0;
        // Compiler error: mismatched types
}
```

Program code 1: Variable bindings

```
fn main(){
    let answer = frac(3,4);
    println!("{}", answer); // 0.75
}
fn frac(numerator: i32, denominator: u32) -> f32{
    if denominator == 0 {
        return 0.0;  // explicit return statement
    }
    numerator as f32 / denominator as f32 // implicit return
}
```

Program code 2: Functions with return values

[15] Rust uses braces ({}) instead of the more commonly used percentage symbol (%).
[16] There is no function overloading in Rust.

2.3.3 Pattern matching

This feature is known quite well from languages like Haskell. It combines a condition with a variable binding in one token. Typically this makes code not only shorter, but also more readable.
The semantics of pattern matching are quite intuitive and will generally not need much clarification. However, there is a number of ways to match patterns in Rust and I will mention the most important among them quickly.
The **match** keyword can be thought of as a stronger version of C's **switch-case**. It is followed by a value that is being pattern matched and different cases follow afterwards. Some examples are shown in code 3, but the possibility are even more complex.[17]
Another way of pattern matching is via a normal **let** statement. More syntactical sugar is provided with the **if let** construct, which makes pattern matching on a single condition shorter than the corresponding match expression.

```
// simple pattern matching on constants
match 2 {
        0 => println!("Nothing"),
        1 => println!("A single one"),
        // We can use ranges with 3 dots, including on both sides
        2 ... 9 => println!("Between 2 and 9"),
        // With the 2 we can bind the value to a variable name
        // When patterns overlap, only the first match is taken
        n @ 2 ... 99 => println!("Exactly {}", n),
        // We can give wild cards with the underscore
        _ => println!("A hundred or more!")
}
// prints: Between 2 and 9

// pattern matching on tuples
match ("Coordinate", 3, 4) {
        // x and y act as wild cards but also bind the value
        ("Coordinate", x, y) =>
                println!("We got a coordinate ({}|{}).", x ,y),
        // Another use of wild cards without binding
        ("Unknown location", _, _) =>
                println!("Unknown location"),
        _ => println!("Incorrect input")
}
// prints: We got a coordinate (3|4).
```

Program code 3: The match syntax

[17] For instance, there can be guards on boolean expressions. Or we can bind values by reference.

2.3.4 The Ownership Model

This topic, the ownership model that Rust uses, is arguably the most unique and defining aspect of the language. In the Rust book, there are three chapters dedicated to explaining the ownership system, they are called *"Ownership"*, *"References and Borrowing"*, and *"Lifetimes"*. Explaining this here in detail would be too long and it is not strictly necessary to understand the code examples. However, to understand the benefits of Rust and to get an idea how it works, it is important to understand what this ownership model is capable of.

The borderline is, Rust guarantees, at compile time, that no data races occur and that there are no memory leaks. Further, any valid pointer will always point to a valid data location, in particular, there are no null pointers. The idea is that every valid memory location, at any given point in time, has exactly one valid write-pointer to it or no write-pointer and a positive number of valid read-pointers. When the last pointer referencing a data location is invalidated or goes out of scope in the source code, then the data is being *freed*, hence no data leaks can ever happen.

To reason about those pointers, each data location has a conceptional owner. Conceptional means, there is no actual ownership stored and checked at runtime. The entire ownership model is only a concept to proof some guarantees at compile time and has no performance impact at runtime.

Borrowing is the term to take the ownership temporarily from the previous owner and returning it afterward. *Lifetimes* are a language feature which is sometimes needed to express specifically how long a data location must be valid. In many cases, the compiler can figure it out automatically and programmers need not worry about lifetimes. But in some cases, the code is ambiguous and only lifetimes can express the exact semantics of our code. In those cases, the compiler will not accept the code without lifetime specifiers, because it can't proof that the ownership model is satisfied without those specifiers.

Doing all of that at compile time is a tricky task. The fact that the Rust community has successfully done exactly that, is probably the main reason, why people get interested in the language.

The downside of this model is, it takes some time to understand it fully and some programs require more thinking by the programmer when written in Rust than it would when using another language.

Generally speaking, I would argue it is a good thing when programmers are forced to think more about their code. The Rust compiler pointed out to me several flaws in first sketches of the data layout in the Rust projects that I have been working on so far. This is very beneficial, as a lot of time can be saved when detecting fundamental problems in an earlier stage of the development process. But of course, there are also examples[18], where the model seems to limit the programmer's freedom and it takes additional effort to convince the compiler that the code is fine when it seems to be trivially true to the programmer that there is no problem with the code. I will talk a bit more about this in section 4.2.3.

A case study on how the driver handles DMAable memory in section 3.4.1 shows some consequences of the ownership model.

[18] Think about a doubly linked list: It naturally stores at least two writeable pointers to each element, which is against the basic rules of the ownership model. While one can certainly work around this, it is not really trivial to do that in Rust.

2.3.5 Structs and Methods

Structs in Rust are pretty much the same as in C and should not need much clarification. The only major difference is that each field is private by default and only accessible from the outside when defined as public, using the **pub** keyword.
However, Rust brings a nice syntax for functions that belong logically to an instance of a struct. This concept is better known as methods in OOP.
Any struct in Rust can have a set of methods that operate on an instance of that struct.[19]
To implement methods for a struct A, a block **impl A** is used. Inside of it, defined functions can take a special **self** argument that refers to the instance of the struct. Again, it can be bound as mutable or immutable, in the latter case the function is not allowed to change any fields. An example of a struct and the method syntax is available in listing 4.
In listing 4, a struct is instantiated directly, but that is only possible if all struct fields are public. In real code, we almost always want to provide at least one *constructor* method for the struct. Constructors do not have the *self* argument which means they are not bound to an object,[20] but they return an object. An example is available in code block 5

2.3.6 Pointers

Rust has more than one pointer type. The different semantics for them is a fundamental tool for Rust to reason about memory. The simplest among them are **&** references, created when borrowing values. They point to stack variables and are guaranteed to be valid.
To allocate memory on the heap, a struct called **Box** is used. It manages the allocation and deallocation of the memory region, guarantees safe memory access and grants a clear distinction between heap and stack resources.
The last type of pointer I want to mention is the **raw pointer**. In most cases, they behave like ordinary pointers in C. Of course that means, in contrast to the two previous pointer types, raw pointers do not have any guarantees on where they point to and they can also be null pointers. Normal pointer arithmetic can be performed with raw pointers and with the *offset* method it can also be accesses like an array.
When a raw pointer is dereferenced, it has to be marked as unsafe code, which will be explained further soon.
To make best use of Rust's type system, raw pointers should be avoided whenever possible, but they are also a powerful and important language feature. Especially in systems code, they are unavoidable. But not much of Rust's benefits are lost when programmers think carefully about how to choose abstractions. A good approach to use raw pointers is to hide them behind some safe interface. An example showing how I modelled a safe interface for a physically mapped memory region is provided in section 3.4.1.
All three pointer types presented can be dereferenced using the ***** operator. When using the method syntax, dereferencing can usually be omitted on the object because of a language feature called *deref coercions*.
For both raw pointers and borrowed references, there is also a distinction between mutable and immutable pointers.[21]

[19] For simplicity, I will also call those instances objects, even though we are not in a object oriented environment.

[20] Compare it with static functions in Java.

[21] Note: Binding a pointer mutably and having a mutable pointer are two different things. The first means that the variable holding an address can be modified, the second means the data region pointed to can be modified. Combing the two is also possible and looks like this: *let mut a = &mut b*

```rust
struct Person {
    pub age: u8,
    pub money: u32, // this last comma is optional in Rust
}
// Implementing the methods for the struct Person
impl Person {
    // borrowing the 'object' mutably
    pub fn has_birthday(&mut self){
        self.age += 1;
        self.money += 100;
    }
}
fn main(){
    // James is modified later, inside the has_birthday method
    // therfore it must be decalared as mutable
    let mut james = Person {age: 0, money: 0};

    // A for loop in rust using the iterator paradigm
    // 1 is included 80 is not included
    for i in 1..80 {
        james.has_birthday();
    }

    println!("James is {} years old and owns {}CHF.",
                james.age,
                james.money
        );
    // prints: James is 79 years old and owns 7900CHF.
}
```

Program code 4: Defining, instantiating and using structs

2.3.7 Unsafe Code

It is not always possible to satisfy all of the strong guarantees of the model Rust tries to follow. Sometimes when we want to access a register of a device, we simply cannot be sure whether the data in it will be valid or not. In systems programming, there are many such examples where this entire ownership model seems to be too restrictive, yet Rust aims to be used in exactly that area.
To weaken some of the constraints of the language, code can be surrounded by an **unsafe** block. The code inside is considered to be *unsafe* and is allowed to break some of the rules. For instance, we can dereference a pointer that we calculated with arbitrary pointer arithmetic. There is no way we could guarantee at compile time that this location will be valid, but as a programmer we can maybe even proof that it is. By marking it as *unsafe*, we tell the compiler that we know what we do and that we think the dereferencing is okay. That doesn't mean we completely turn off the type system of Rust, in fact most rules still apply. The two abilities to dereference raw pointers and to call other unsafe functions already sums

```rust
// Implementing the constructor
impl Person {
    pub fn new(age: u8) -> Person {
        // creating the object
        Person {
            age: age,
            money: 100 * age as u32
        }
    }
}
fn main(){
        // Calling the constructor.
        let john = Person::new(79);
        println!("John is {} years old and owns {}CHF.",
                john.age,
                john.money
        );
        // prints: John is 79 years old and owns 7900CHF.
}
```

Program code 5: A constructor in Rust

it up pretty much, what an unsafe block allows programmers to do. The chapter *Unsafe* in the Rust book explains this in more details and also talks about what guarantees are still achieved.

2.3.8 Inline Assembly Code

Systems programmers sometimes need the ability to escape the language completely and write hand crafted assembly code. Maybe a specific instruction has to be called to implement a lock or maybe the performance of the compiled binary is just not good enough at a specific point. In any case, systems programming is unthinkable without the possibility to write inline-assembly. Therefore the Rust community has implemented the macro **asm!** which allows exactly that. At this point, I only want to mention that it is possible to write such code and refer to the Rust book once again for the detailed documentation.

2.3.9 FFI

By the time of writing, C is considered the lingua franca of systems programming. It is not surprising that the Rust community has put a lot of effort into making the foreign function interface (FFI) towards C code and backwards as smooth as possible.
To develop the NIC driver, I often had to call functions from Barrelfish's libraries, for instance to connect up to the network stack. This can be done by simply declaring the function with the correct signature somewhere in a **extern** block and then calling this function. Data structures defined in C must be defined again in Rust. A short annotation as seen in example code 6 makes sure the alignment is following the C standard.

```
// Functions declared here are defined somewhere in C code
//  and can be called from Rust code
extern "C" {
        // A simple example for a foreign function definition
    pub fn pci_client_connect() -> Errval;
        // A more comlpex example that shows how to make callbacks
        // The first argument is a function handler
        pub fn pci_register_driver_noirq(
                init: pci_driver_init_fn, class: u32,
                subclass: u32, prog_if: u32, vendor: u32,
                device: u32, bus: u32, dev: u32, fun: u32
        ) -> Errval;
}
// Defining the type of the callback function above
// Extern means the function can be called from foreign code
pub type pci_driver_init_fn =
        extern fn(bar_info: *mut DeviceMem, nr_mapped_bars: i32);

// repr(C) makes the struct's internal alignment C standard
#[repr(C)]
pub struct DeviceMem {
   /* Some fields*/
}

// The callback function to be called from C code
extern fn init_driver(
        bar_info: *mut DeviceMem,
        nr_mapped_bars: i32) {
        /* Some function body */
}

pub fn main() {
  // Calling the foreign function requires an unsafe block
  unsafe {
    pci_register_driver_noirq(init_driver, 0);
  }
}
```

Program code 6: Calling a C function from inside the barrelfish PCI library

2.3.10 Traits

It is not crucial to understand traits completely to follow the code samples I present in section 3, but I will mention traits several times and they form one of the high-level features that makes Rust really distinct from languages like C. Because of that, I will give a rough overview of the idea behind traits and how they work.
One of the most fundamental features of OOP languages is subtyping, often accomplished

through inheritance. This is a useful concept to allow polymorphic functions.
In Rust polymorphism is achieved using *traits*, working in a similar fashion as type classes in Haskell. We can define a trait with a set of method signatures, either with or without a body. Then any struct can decide to implement that trait, which requires a function body for all method signatures in the trait without implementation.
In a next step, another function can define its input parameters to be valid if and only if the object's type implements that trait. In the function body, this will allow the use of all methods defined in the trait definition.
This type of polymorphism is a great tool for many applications and is highly performance-optimized by the Rust compiler. Whenever possible, the functions are dispatched statically such that it has no impact on the runtime. This is one of the concepts that the Rust community likes to describe as zero cost abstractions. On the other hand, if necessary, Rust is also capable of dispatching methods dynamically where necessary.
As a side note, traits are not as flexible as the inheritance in C++ for instance and they require a slightly different perspective when using them. Taking a finished implementation from an OOP-language of choice and translating the inheritance structure directly to traits will not always work out. Indeed the concepts are quite different, but they usually serve the same purpose.

3 Implementation

The main focus of this thesis is about exploring the feasibility of Rust as a systems programing language. The purpose of this chapter is to elaborate how Rust made the development experience different. The structure is arranged such that language features are in the center, and code samples are primarily selected to show the differences between Rust and C.
About the functionalities of the card driver itself, I will only cover the key ideas and not go into details of how everything is implemented.

3.1 Versioning issues

The development branch of Barrelfish changes in a daily basis and most student projects are working on one specific version of it and are not updated to new releases of the OS. Because of that, the Rust port for Barrelfish[9] was only available on an old version of Barrelfish and I had quite a few errors due to conflicting versions. The reference driver implementation written in C used an even older version of Rust and has not been updated since the project was finished in 2015.
As part of my project, I have integrated both projects to an up-to-date development branch. This involved minor changes to the Mackerel compiler, the Hake building system, and the Rust interface towards some user-level libraries and towards capabilities.
I have spent a lot of time just to make everything function together as required. This is a typical issue that appears when using a non-standard tool to work on an existing project. Thus I mention it here as a drawback for using Rust inside of a large project, like Barrelfish, that has mostly been developed in C. But of course versioning issues are not a problem exclusive to new programming languages and can be encountered in many different forms.

3.2 Building with Cargo and Hake

Getting a simple application to run on a machine with the correct hardware and running Barrelfish on it involves a number of steps. Therefore there are many potential points of failure. Given the different versions as discussed in the previous section, many of those turned into actual failures and I had to dig deeply into the building process to resolve them. For example, there have been some changes to Hake in the end of 2016 which have not taken Rust into consideration. Thus I was looking at the generated Make-rules and modified the corresponding Haskell code of Hake[22], to fix the problems.
Building Rust on a mainstream OSs typically works with the package manager called Cargo[23]. It is similar to Ruby's package manager RubyGems in its functionality and lets programmers define some dependencies to libraries in a meta data file.[24] All referenced resources will then be downloaded and linked automatically when Cargo is invoked.
Cargo is a great tool to improve the productivity of developers. It also manages different versions of the same library smartly. Given a project with dependencies to different libraries which depend themselves on other libraries, it might happen that some of our dependencies use different versions of the same underlying library. Cargo will hide all these complications from the programmer and possibly saves him or her a lot of time here.

[22] see section 2.1.3
[23] https://crates.io/
[24] The files are named Cargo.toml and are written in *Tom's Obvious, Minimal Language (TOML)*

In the context of Barrelfish, Cargo is not in use. Instead, Hake invokes the Rust compiler directly, partially because Cargo did not offer many functionalities to be integrated into other build systems by the time Rust has been ported to Barrelfish.[9]

3.3 Writing Low Level Systems Code

When I first heard about the Rust's safety guarantees, I thought it will probably be very hard to write low level code in that language. Much of the C code I have seen before seemed to be somewhat hacky and I couldn't imagine how this works together with those guarantees.
Starting to use Rust, I could see that its community has a good understanding of what kind of code can be written in a safe way and what not. Having the safety restriction on all the code would make systems programming impossible. But as it turns out, breaking the safety rules in a small fraction of the code base is enough if all abstractions are chosen wisely. Marking that part of the code as *unsafe*[25] and careful reasoning about what happens inside, allows to hide the unsafe actions behind a safe interface. In this way, programmers can have the full benefits of the safety features in the rest of the code base. But even in the unsafe part, most typing rules of Rust still apply.
The next few subsections discuss different forms of typical low level code and how it can be done in Rust code.

3.3.1 Register Access

One typical occurrence of low level code in a hardware driver is a register read or write. Because that requires to access single bit fields at specific locations in address space, the question arises, how well this can be represented in Rust. The answer is, it is just as easy as with C. Of course we need a code block marked as *unsafe* at the very low level, but otherwise the operations look exactly as they would in C. As a plus, we can rely on the type checker of Rust to enforce the register length matches the provided value to be written. On the other hand, the address calculation is just as error prone as in C, the only difference is that we can immediately see the danger because of the enforced *unsafe* block.
The heavy lifting for register reads and writes on Barrelfish is done by Mackerel.[26] Because of the Rust port to Barrelfish[9], Mackerel can also generate Rust code which provides a *safe* interface to us as driver developer. Once all registers and bitfileds have been described in a Mackerel file, a register read or write is simply a matter of calling a method on the corresponding Mackerel device struct, as shown in code 8.

3.3.2 System Calls

The allocation of memory that allows for DMA is required in NIC driver, therefore it needs to know about the *physical address* of the allocated memory. This is an example where a direct *System Call* is needed.
One difficulty is that a *System Call* has different implementations for different architectures and it requires inline assembly code to be performed. Further, the number of actually used arguments varies for different types of *System Calls*. The unused arguments are typically

[25] see section2.3.7
[26] See section 2.1.6

```
// A function to write to a specific bitfield of a 32-bit register
pub fn pcistate_bus_mode_write(base: u32, val: u8) {
  // The bitfield of interest is at position 2
  let mut register_value = ((val as u32) << 2) & 0x4;
  // 112 is the offset of the register
  let addr = base + 112;
  register_value |=
    // volatile read requires unsafe block
  unsafe {
     intrinsics::volatile_load(addr as *const u32)
  };
  // volatile write requires unsafe block
  unsafe {
    intrinsics::volatile_store(addr as *mut u32, register_value);
  }
}
```

Program code 7: Low level manipulation of registers and bitfields

```
fn clear_some_flags(d: bcm5704::Device) {
    // clear flags by writing 1 to the corresponding bitfields
    d.ap_status_err_write(1);
    d.rx_risc_state_halt_instr_write(1);
    d.rx_risc_state_blck_read_write(1);
    d.tx_risc_state_blck_read_write(1);
}
```

Program code 8: Calling the generated functions of Mackerel

all set to zero. But having to write about ten redundant zero arguments in a function call is clearly undesirable.
The solution implemented in Barrelfish uses a recursively defined macro to fill up missing arguments with zero values. The inline assembly code is not a problem either, as discussed in section 2.3.8. With the Rust port to Barrelfish, all of this is hidden behind a short macro call that performs the system call, as shown in the code example 9.

3.3.3 Device Data Structures

Hardware devices usually come with some definitions of data structure they are using. Often these structures have to be represented in the driver software. It is vital, that the memory layout is exactly as required. This is where the FFI in Rust comes in very handy: a simple annotation before the struct can define the struct to be laid out in a way expected from a C compiler. An example is available in the code listing 6 in section 2.3.9.
As seen in section 2.2.4, the brodacom 5704C chip is no exception and also defines its own data structures. Obviously, a Rust code representation for all of them was necessary. But the alignment of the fields is not the only thing to consider. There are other issues to be

```
/// Allocates a DMAable frame of memory and
/// returns its physical and virtual address
dma_frame_alloc(size: usize) ->
    Result<(*mut libc::c_void, u64), ErrorCode> {
  let mut frame_cap = Capref::new();
  let mut frame_id = frame_identity{base: 0, bytes: 0};
  let mut allocated = 0;
  unsafe {
    // Error handling omitted
    // A function provided by Barrelfish to get a capability
    frame_alloc(&mut frame_cap, size, &mut allocated);
    // The syscall hidden behind a macro
    invoke!(
      frame_cap,
      frame_cmd::Identify as usize,
      &mut frame_id as *mut frame_identity
    );
  }
   /* Mapping into virtual address space and
      returning the addresses omitted */
}
```

Program code 9: A system call in Barrelfish

careful about, one that I want to discuss here is the byte ordering.
The chip internally uses a 64-bit big-endian representation, but the PCI standard is little-endian. So when the driver receives an 8 byte value over the PCI bus, the chip will remap the bytes automatically as shown in figure 6. But many registers are only 4 bytes wide and the remapping will also apply to those values, thereby accessing the wrong location. To ease the access of 4 byte values, the chip comes with a configurable bit for word-swapping. If enabled, the mapping is redefined to work for 4 byte reads and writes, basically just a reordering of the higher and lower four byte words.
To pick just one example, what does activating this word-swapping option mean for accessing RCBs[27]? It means that the fields can be accessed normally and everything works as expected, except for the 64-bit physcal address stored in the first field. The address will be word swapped when it is written to the chip's internal memory, thus the swapping must be negated. When manually swapping the words each time the values is accessed somewhere in the code, I argue this leads to confusing and error prone code.I want to suggest something different.
With the notion of private struct fields in Rust, a restriction on how fields of the RCB are accessed can easily be realized. In that way, the internal structure of the RCB is completely hidden to code outside of the RCB's methods and the complication can be isolated within the module. How I ended up storing the fields is shown in listing 10. Note that the constructor takes a 64-bit address in normal representation, meaning that all word swapping is hidden completely behind the interface of the struct.
This fiddling around with byte and word order is not uncommon in systems programming.

[27] See section 2.2.4.

On the BCM570C chip are a total of six configurable bits to set up different modes of swapping bytes and words in different contexts, all of them need to be set up correctly by the driver.

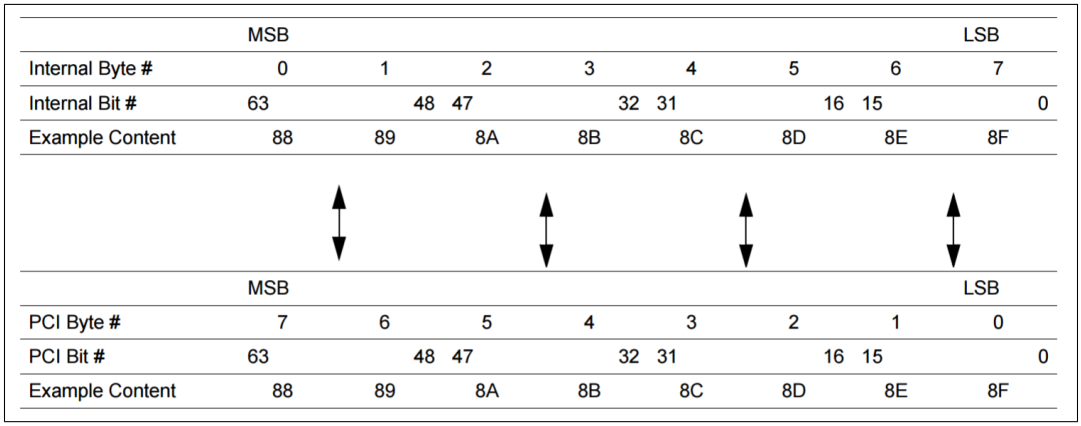

Figure 6: Byte reordering taken from figure 91 in the programmer's guide

```rust
/// The ring control block struct.
/// It defines where a buffer descriptor ring
/// is stored. The max_len field can have different semantics,
/// depending on the type of the ring.
#[repr(C, packed)]
#[derive(Debug)]
pub struct BcmRcb{
    host_addr_hi: u32,
    host_addr_lo: u32,
    max_len: u16,
    flags: u16,
    nic_ring_addr: u32,
}
impl BcmRcb {
    /// Creates a simple instance of the struct.
    // Be careful about the u64 address value here!
        // The broadcom devices uses BIG endian internally
        // and we use the word swapping configurration for
        // little endian hosts,
    // we need to store the HI 32 bit at the lower address
    pub fn new(host_address: u64, len: u16, nic_ring_address: u32)
            -> BcmRcb {
      let hi = (host_address >> 32) & 0x00000000FFFFFFFF;
      let lo = host_address & 0x00000000FFFFFFFF;
      BcmRcb{
            host_addr_hi: hi as u32,
                host_addr_lo: lo as u32,
        flags: 0,
                max_len: len,
        nic_ring_addr: nic_ring_address
          }
    }
}
```

Program code 10: The RCB structure represented in Rust code

3.4 General Data Structures in Rust

3.4.1 References as fields

A characteristic difference between C and Rust is notable when a reference should be stored as a struct field. A direct translation from such a C struct to Rust might be rejected by the Rust compiler because of lifetime errors. Sometimes that means that there is a missing lifetime specifier, and sometimes it is a signal that there is something conflicting the guarantees Rust tries to accomplish. The first case is not really interesting and can be fixed easily, but the latter is worth discussing. This case can be frustrating for new programmers since no addition of lifetime specifiers will resolve the lifetime error thrown by the compiler. Instead, the design of the structure must be modified.

To get some intuition about references stored as struct fields, different ideas will be discussed here.
The crucial question is, what type of pointer should be used to store a reference in a struct. Storing normal &-references forces the compiler to prove that the reference always points to a valid address, which the programmer has to make possible. First, the underlying memory location must be initialized right away when creating the instance of the struct. Or alternatively, the reference could be wrapped it in an $Option< T >$ to make it explicit that it can be uninitialized.
But then, no matter how or when the value is initialized, there might be some lifetime problem since the referenced memory must be valid at least as long as the object is. This cannot work in the constructor because local variables of the constructor will be invalidated as soon as the constructor terminates. Doing it outside of the constructor will lead to similar problems, unless the data is allocated on the heap. It seems that a normal reference is not the right choice, which makes sense since it is for stack allocated memory.
So the obvious answer is that a Box type for heap allocated memory should be used, unless there are good reasons to store references to stack values. If that is actually the case, then the lifetime specifiers are the perfect tool to detect memory safety problems which are easily missed in C or C++, where the field could point to an invalid stack location without any compiler errors.
But for systems programmers, there is more than just the heap and the stack. An example from the implemented driver is the struct representing a ring as described in section 2.2.4. It stores a reference to the buffer descriptor array in the field called *ring*, which has to be accessible through DMA as well. Therefore, allocation happens through a syscall and will not yield a Box object.
With Rust's standard library, it is possible to create a Box from a raw pointer. But that will bring some hidden problems, as the Box object manages the underlying memory and therefore it will *free* the memory when it is invalidated. If that happens to the memory allocated through the syscall, undefined behavior occurs because memory is freed that has not been allocated with the Box' allocator.
Giving up the idea of using a Box, the only viable choice to store the reference are raw pointers. Thereby, compile-time safety is given up to some degree. But with the right abstractions, this issue can be kept isolated and should not affect the rest of the code base.
My solution to this exact problem was to create a new struct managing the underlying memory allocation. I called that struct DmaArray. It has an interface like an array and its purpose is to manage memory on which the driver needs direct memory access.
All unsafe code for allocating the data is done inside methods which themselves form a safe interface. Nobody except the DmaBox will deallocate the memory location and the amount of allocated memory is remembered. Each access is boundary first, making sure our raw pointer dereferences access only the allocated memory region.
My implementation is shown in code block 9.2.1 of the appendix.
Because my struct implements the *Index* trait, the use of the DmaArray is just like a normal array, as shown in code block 11.

```
let size = 100;
// Allocate the memory
if let mut dma_able_array = DmaArray::<f64>::new(size) {
        // access data like an array
        dma_able_array[4] = 4.2342;
        assert_eq!(dma_able_array[4], 4.2342);
        // That would panic at runtime:
        // dma_able_array[200] = 0.0;
}
else { /* Handle memory allocation failure */ }
```

Program code 11: Creating and accessing a DmaArray

In the driver, the DmaArray is not used to store floating point primitives as in the example code, but it holds the three different types of buffer descriptors. That brings up some questions about the ownership of non-copyable elements within the array.
When an object is inserted, the DmaArray takes the ownership of that value to prohibit aliasing. When the value is read, usually it will be borrowed and the ownership stays at the DmaArray. But when an element is removed, the DmaArray gives up the ownership of it. To ensure further reads to the same index will not cause aliasing, I decided to effectively copy the entry upon removal and then zeroing the memory location. If that array slot is read again later, it will be empty.
The current semantics of the DmaArray are that empty slots are just represented as zeros and the programmer has to make sure no invalid locations are accessed. If a user wants to be guaranteed at compile-time that each access on the field will read a valid instance, Option$< T >$ could be used to differentiate between empty and full slots.[28]
Peeking beyond the scope of my own project, I think applying the ownership model to DMAable memory regions like that is a good starting point to prove the correctness of systems like a driver.
Going back to the problem with pointers stored in fields, how did this solve our problem? Instead of storing a reference, a DmaArray can be stored as field of a struct. I argue that this is a good abstraction because it is pretty intuitive, easy to use and most importantly, it tells a true story. When looking at the ring struct, a DmaArray is stored, thus it is visiable that the memory is handled in a different way. With any standard pointer type , this fact would be completely hidden.
Looking back to the statement that doing such things in C is typically different from what people do in Rust, I must stress that Rust does not actually make it harder for us. There is always the possibility to use raw pointers directly and wrapping everything in unsafe code. But this is bad style and is very error prone. Bad style and error prone in Rust, but just as much in C as well. When storing a pointer in C, no compiler error shows up. Will programmers always know that it is pointing to a special memory location? Who is deallocating the memory and when? How easy is it to argue about the correctness of the code after it is all written down? Or one year later, when some other programmer has modified the code?

[28] Note that the DmaArray will also be accessed by hardware and the underlying memory can change at any time. Therefore the driver is not in complete control of the memory and using an Option type here would promise safety which cannot really be guaranteed by host software alone. Zeroing makes it so that all values inside are forgotten even by hardware.

I think the wrapper struct helps to deal with these issues. So I would also suggest using this design when programming in C. The syntax will not quite work as easily, but we can make the same abstraction. The real question is, would I actually do it this way in C? Probably now I would, because I can see the problematics, but in other examples, I will maybe miss it. Problems like the one with the DmaArray are quite common in systems programming and C does not show them well. Rust on the other hand often points the finger right on the spot.

3.4.2 Private fields and methods

In OOP it is generally considered a bad practice to have many public fields in a class. There are many reasons for that and most of them also apply to other programming languages. A big one is that code is hard to reason about. There is no easy way to enforce an invariant on a public field. Another one is modularity and closely related, backward compatibility. Basically, when we have a public field on a structure that some other module might access, then there is a dependency between the modules. That is bad if the implementation of the struct changes. If the field was private in the first place, we would have the guarantee that all code has to access it through methods. Having all those methods in one single file makes it a lot easier to reason about code, and it produces better isolation between modules.
In my code of the NIC driver, I thought carefully about the access modifiers for my struct's fields. Most of them are private, sometimes with typical, one-line getter- and setter-methods. Marking them for enforced inlining helps to dispel any skepticism about performance. But in many cases, there are no such methods needed because all operations on the fields happen inside of methods anyway. Having the right abstractions should essentially always facilitate that.
There are two exceptions where I was still using public fields. The first one is for my top level struct called *Driver*. It holds the entire context of a driver instance and is used in all callback functions provided to the NQM. Because each of those callbacks operates directly on variables of the Driver struct, they are semantically like methods. Putting all the code into actual methods would have forced me to write callback functions which do nothing besides calling the method on the globally stored struct, which seems like an unnecessary indirection to me. Instead, I made many of the fields public and thereby not giving up that much, since no other module will ever want to instantiate that struct and inside my module, it is the top most struct anyway.
The second reason I used public fields is debugging. I wanted to write some functions that print the state of a number of structs, also including fields that should not be accessible from outside for productive code, thus writing getter-methods is not my first choice. Rust has a nice debug trait, discussed in section 3.6, which helps greatly with the printing part. But one of my debugging functions was also reading the fields to decide what else should be printed. To allow this, I was making some fields temporarily public, as long as I was debugging.

3.5 Bringing Everything Together

When it came to writing the most the highest-level code of my driver, it felt quite a bit different than the details mentioned in previous sections. In the main function and in the callbacks for Barrelfish's NQM, I was able to use the safe interface of my other structs.
In a perfect world, there would be no unsafe code in those functions. The main reason

why I required unsafe blocks is caused by an unfortunate implementation of the NQM. Its callbacks don't support any way to pass a context along which leaves programmers with no other choices but using a global state.
Global state that can only be read is generally no issue, but we are looking at mutable global state. This obviously comes with many dangers of inconsistencies when different threads or processes access it, so in Rust any access on it is unsafe.
In my implementation, it's relatively simple to reason that the global access is consistent, because there is only one single thread accessing it, but still, I would like to avoid using global state. Losing modularity and isolation in that way is undesired and it makes it harder to argue about invariants. But that problem was not really hindering me to program what I wanted, it's just a small blemish to have that additional unsafe keyword at the start of each of those functions. And of course, I also used unsafe blocks whenever calling functions from Barrelfish libraries directly.
Besides that, it was really nice to write the code for the main and callback functions. In contrast to the C code I had as a comparison, I was able to call constructors and methods. Further, I did not have to worry about memory deallocation. Thus, the general experience was like being in an object oriented environment with a garbage collector, which of course, Rust does not have.
Error handling is also visibly cleaner and overall the object-method abstraction makes the code more intuitive. I really came to appreciate that when I extended the finished driver with an Internet Control Message Protocol (ICMP) manager that I implemented to answer pings directly inside the driver. The ICMP manager needs to send data without receiving allocated buffers from the NQM, therefore it must allocate DMAable memory on its own. Clearly, the data should not be allocated each time a ping is answered, therefore the need for yet another ring structure arose.
With a bad code design, it might be a cumbersome task to allocate the physically mapped memory, handling the access to it, capturing all possible errors and so on. In my case, I the DmaArray module encapsulated all of that already. The memory allocation became a single line and access to the memory location is immediately given through the array-index syntax (rectangular brackets [ ]). The physical address that the NIC requires is available through a getter method *get_physical_address()*.
The full constructor for the ICMP manager is shown in listing 12. The *try!()* macro unwraps the value returned by the memory allocation if successful and returns the error otherwise.

```
pub struct IcmpManager {
    buffer_pool: Vec<DmaArray<u8>>,
    producer_index: usize,
    consumer_index: usize,
}
impl IcmpManager{
pub fn new(size: usize, config: &Config) -> Result<IcmpManager, ErrorCode> {
    let mut buffers = Vec::with_capacity(size);
    for i in 0..size {
        buffers.push(
      try!(DmaArray::new(config.get_max_frame_size() as usize))
    );
    }
    Ok(IcmpManager {
         buffer_pool: buffers,
         producer_index: size - 1,
         consumer_index: 0,
  })
}
```

Program code 12: Reusing the DMA module for the ICMP manager.

3.6 Debugging

Debugging is a significant aspect of software development. It often takes a substantial amount of the resources available for a project, so it is important to know the tools at hand. I will start with the most basic debugging tool: Printing output into the console. Very useful for virtually any project and used during the entire development process. But it goes much further than just printing out a message like *Reached checkpoint 5*, debugging output is often representing the state of structures and conditional on many other structs.
At this point, I want to mention the *Debug* trait that does ease printing the state of structs a lot.
When a struct implements the Debug trait, it can define exactly how the internal state should be represented in a string to output it to the console. To actually print it, an object can be passed as a parameter to the print function and the debug representation can be selected with a **:?** modifier, as shown in listing 13. In some way, Java does the same with the *to_string()* method. But Rust facilitates different string representations, there is also a *Display* trait that is used for non-debug prints.
Very handy in this context is the possibility to derive traits automatically. A short attribute at the struct definition is enough to generate code that prints all fields' names and values. Derivation is also possible for some other structs of the standard library and with recent changes programmers can even implement auto-derivation for custom traits.
Debugging with other tools than console output is often desired, for example when strange bugs occur. In that case, it is possible to use common debugging tools like GDB since Rust generates native debugging symbols. Being able to see all variables change dynamically at runtime and setting up breakpoints makes debugging so much more productive.
For me, working on Barrelfish, setting up GDB to work would have been a tricky task and

therefore I stayed with simple console output for debugging.

```
#[derive(Debug)]
pub struct BcmStatus{
    block: DmaBox<BcmStatusBlock>,
    frame: u64,
}
// somewhere in the driver module
let status = self.status;        // immutable binding is enough
println!("Status block: {:?}", status);
        // Because DmaBox implements the Debug trait as well,
        // this will print the frame value, but also all fields
        // and values of the BcmStatusBlock, even though
        // the fields are private
/* Output */
Status: BcmStatus {
block: DMA Box at va 0xdb685000 = pa 0x134bb000 contains:
        BcmStatusBlock { word: 5, tag: 22, rx_jumbo_idx: 0,
        rx_std_idx: 127, rx_mini_idx: 0, reserved: 0, idx_pair: [
        BcmStatusIdx { rx_prodidx: 639, tx_considx: 5 },
        BcmStatusIdx { rx_prodidx: 0, tx_considx: 0 },
        BcmStatusIdx { rx_prodidx: 0, tx_considx: 0 },
        BcmStatusIdx { rx_prodidx: 0, tx_considx: 0 }] },
frame: 323727360 }
```

Program code 13: Automatically deriving the Debug trait and using it in a console print.

4 Evaluation

4.1 Rust as systems programming language

A drawback in productivity that I experienced while using Rust was that the surrounding tools have not been tested and approved in practice.
In my case, I've been using a research OS which is in continuous development as well as a language that is still being extended and stabilized day by day. This combination caused smaller and larger issues over the time, which would not have occurred with the long time tested language C.
For instance, I had to track down some bugs in the Mackerel compiler back-end for Rust, before I could use the auto-generated code for my device. The reason why nobody has noticed the bugs in Mackerel before is simply that nobody has actually used its Rust back-end. When it was implemented, it was only tested with a very basic device which did not cover all of Mackerel's functionalities.
This general issue which shows up when a new tool is used in an old environment is currently one of the largest reasons against using Rust, I think. If existing programming teams should choose to migrate to Rust, it will not be enough when Rust is just as good or slightly better as the language they have used so far. It must be much better for programmers to go through all the effort of learning a new language, learning about the available tools, dealing with the lack of practically approved libraries, and so on.
But seeing the efforts of the Rust community and how it is growing, I think we will see the popularity of Rust growing over time. With that, just mentioned problems will dissolve slowly and Rust could become more and more viable for future projects.
Another reason why I think the Rust has a lot of potential in the future is its focus on multi-threaded applications. The recent state of technology progress seems to suggest that the number of processor in most devices will be more than one. Already now single core home computers, laptop or smart phones are almost extinct. This means for programmers that they must write concurrent code if they want to run fast. This is considerably harder than writing non-concurrent code. Using languages which are not designed for that purpose makes this issue even bigger and developers will seek for solutions.
Rust comes in and offers all tools well integrated into the language that allow implementing safe concurrent code. This is where I see the potential for the language.
Looking at the language isolated from the state of art and available tools, I can say with quite some confidence that Rust is very well suited for systems programming. The language makes everything possible what a systems programmer ever wants to do, as I showed partially in section 3.3. It also brings many useful higher-level abstractions with it whose use I demonstrated in various code samples, particularly in sections 3.4 and 3.5.
The syntax is, at least in my opinion, much clearer and intuitive than most other languages. I will give an example for this in section 4.2.2 showing what I mean exactly.
For above reasons, I conclude that Rust is perfectly viable as a language to be used for systems programming, but it might take some more time for people and tools to catch up completely.

4.2 Rust in direct comparison to C

C is the single most dominant language for a lot of areas of programming. It lets programmers talk in a very direct way to the hardware, more so than any high-level language. Assembly and C are very close to each other, understanding the on helps to understand the

other as well.
Rust distances itself from assembly more than C. More concepts go into the language that are not present in assembly and the focus is to make it intuitive for the programmers and their human way of thinking.
Which language really fits best is always dependent on the context. In this section, I will point out a few differences between the languages and explain why I like which approach better.

4.2.1 Safety

One obvious advantage of Rust are the safety guarantees it provides. For applications where system failures are fatal, often only single threaded systems are considered to avoid additional error sources. But with Rust, it can be much easier to reason that code running in several threads, and possibly many cores, is still correct, whereas in C it seems very hard to make a statement about that kind of behavior.
But the promoted safety feature coming from the ownership model is not the only place where Rust shows an improved safety compared to Rust. The entire syntax and type system is built around problems that have been showing up over and over again in the history of programming.
Not allowing null pointers is one great example for this. Many bugs could be avoided by simply checking a pointer for null before dereferencing it. Rust eliminates that problem from the language completely.[29]
Another example is type conversion, in particular for pointers. In C it is barely visible when a pointer is cast to another. Maybe the address initially points to an integer, but then it is converted to a void pointer when it is passed as a function argument. This function can then cast it to anything it likes and the code will not really look suspicious. Rust addresses this problem and does not allow such conversions, a cast is only valid of the corresponding trait is implemented which ensures a correct conversion. And if we really want to circumvent the type system at some point, then we have to be very clear about it.
Casting a variable from one type to an arbitrary type of the same size in Rust is performed through an unsafe function whose use is very much discouraged in the official documentation.[7] But even if we use that, we have to write down specifically the two types involved and wrap everything in an unsafe block. Thus it is much clearer to see in the code that something potentially dangerous happens. I claim this is a better approach than a simple cast as known from C for virtually any code that will go into productive industry.
But type conversion also happens between primitive number types. In C that is often hidden from the programmer's eyes. When adding an integer to a floating point number, the C compiler will make the necessary conversions implicitly. In Rust, it must be stated explicitly how the conversions should happen. In some cases, this is tedious in Rust and it seems that this is the price we pay for type inference. But it makes the code also much clearer and sometimes helps to prevent bugs. A direct comparison is shown in listings 14 and 15 with a made up example that shows how C code can be confusing in this regards. To make the Rust code behave the same way, some type annotations are required. The otherwise occurring compiler errors are attached as comments.

This explicit mentioning of casts is exactly how it becomes clearer in Rust, even though we often don't write any explicit types in Rust. We can see that inference only applies when there is no ambiguity and therefore always produces the expected result.

[29] At least if we ignore raw pointers for the moment

```c
float f1 = 1.1,  f2 = 3.7,  f3 = -0.3;

int i1 = f1 * f2 * f3;
unsigned int u = f1 * f2 * f3;

int i2 = 1;
i2 *= f1;
i2 *= f2;
i2 *= f3;

printf("%d | %u | %d | %f\n", i1, u, i2, f1 * f2 * f3);
// -1 | 4294967295 | 0 | -1.221000
```

Program code 14: Implicit casts on numerical values in C

```rust
let f1 = 1.1; let f2 = 3.7; let f3 = -0.3;

//let i1 : i32 = f1 * f2 * f3;
//               ^^^^^^^^^^^^
// expected i32, found floating-point variable
let i1 : i32 = (f1 * f2 * f3) as i32;

// let u : u32 = f1 * f2 * f3;
//                     ^^^^^^^^^^^
// expected u32, found floating-point variable
let u : u32 = (f1 * f2 * f3) as u32;

let mut i2 = 1;
// i2 *= f1;
// ^^^^^^^^ the trait `std::ops::MulAssign<{float}>`
// is not implemented for `{integer}`
i2 *= f1 as i32;
i2 *= f2 as i32;
i2 *= f3 as i32;
// Type interfered
let a1 = f1 * f2 * f3;

println!("Explicit types: {} | {} | {} | {}",
                          i1, u, i2, f1*f2*f3);
println!("Interfered type: {}", a1 );
// Explicit types: -1 | 4294967295 | 0 | -1.221
// Interfered type: -1.221
```

Program code 15: Implicit and explicit casts on numerical values in Rust

To mention real examples, in the driver for the BCM5704C written in C, I found a couple of functions taking the wrong input type. For example, a 32-bit integer is declared as an argument in the function signature, but inside the function, it is used to write to a 16-bit field. Implicit type conversation effectively hides this fact from the programmer and it can lead to undesired behavior.
Global variables are also treated with more respect than in C, as described in section 3.5. This helps to avoid another type of bugs, but it is also cumbersome for programmers when they want to write some quick and dirty code. Global structs must be initialized at compile time, which implies a constant value for it must be available. Writing all zeros is an option to make it run, but that is essentially lying to the type system since the struct has not really been initialized properly. So the treatment of global variables might be a drawback for some programmers, while others will be glad to have the type system helping them out to ensure the correct handling of global state.
To sum up the safety discussion between Rust and C, let me finish with a small and slightly exaggerated metaphor: C is a deep forest inside of which bugs can hide easily. A forest is a perfect place for camping, but it is not very well suited for a dentist. Clearly, a dentist likes to work inside a building, where the walls and the roof give some guarantees that certain things will not happen. No birds will fly around her head, no snake will bite her and no bugs will drop from trees. For that, she is ready to give up the freedom she could experience outside in nature.

4.2.2 Syntax

Many languages have some obscure syntax that everyone just accepts and nobody questions it because people are used to it. Examples include a typical C-style for-loop, variable declaration and function pointers. The syntax has probably been chosen like that because it is compact. Once a person is used to it, the syntax is clear anyway, so it is understandable why the compact syntax is preferred over intuitive syntax.
Rust diverges and tries to make the syntax closer to the human way of thinking, further away from machine code. As an example, humans often think about variables as a general placeholder for some value. The variable can have different types, but that is mostly important for the machine, not for the human. Then there are also functions, which are very different from variables in both the human mind and the representation on the machine. Functions also have a type and they can produce values, but at its core, it is a set of instructions to be executed and has nothing to do with a variable.
In Rust, there is one keyword to introduce any kind of variables and one for functions, **let** and **fn**. Both are independent of types which are declared in other ways. In C, a function declaration and a variable declaration look very similar and the exact same keywords are involved. The only way to tell the difference is by looking at the brackets, but even that can be very confusing for people who have not used such languages before. Listing 16 shows what I consider to be confusing code.

Another example where C-like syntax is reusing the brackets instead of introducing a new keyword is in type-casting. Rust uses the keyword **as** to make a cast instead.
Rust has built tuples right into the language. That also allows for a nice syntax when a function returns multiple values, exemplified in listing 17.
In C, the same can only be done when defining a new struct just for that purpose and compilers might find it difficult to inline that efficiently. Instead, additional parameters are used in C when a function produces more than one output value. Those parameters

```c
// Note: Each line is valid C99 code
int Int; // variable
int *Int; // pointer variable
int Int(); // function
int (*Int)(); // function pointer
int Int(int Int); // function with parameter
int (*Int)(int); // function pointer with parameter
```

Program code 16: Confusing syntax in C

```rust
fn receive() -> (Box<[u8]>, usize);
fn main(){
    let (data, size) = receive();
    for i in 0..size {
            println!("{}", data[i]);
    }
}
```

Program code 17: Multiple return values with tuples

are pointers to the location where the function should write the value to. That approach breaks modularity and isolation between files, but there is not really a better solution to that problem in C.
An advantage of C and its syntax is easy operating on pointers. I am specifically thinking about the keyword **sizeof** and the macro **offsetof**. Both help when calculating an address of, let's say some field in an array of structs. Rust does currently not have direct support for the latter and the former is done by a function of the standard library. The problem is that this function, at the time of writing, is not marked as *constant*. Therefore it is not a compile-time constant which can have quite some impact on its utility.
Both problems mentioned can be worked around in Rust, but it is more effort than necessary. Therefore C is clearly more useful for address calculations where those operators are used, compared with Rust and its current state of the standard library.
While writing the code to initialize the BCM5704C, I realized that this kind of pointer arithmetics can often be avoided completely. Because I translated most of the C code quite directly in the first place, I also used my own **offsetof** macro in Rust. But later on, I removed it again since I did not like the code design. The relevant code is available in the appendix, listing 21 shows how the RCB was copied into the card memory in the existing C driver, my own implementation is in listing 22

4.2.3 Limitations caused by the Ownership Model

In section 2.3.4 I mentioned that there are examples when the ownership model is hindering programmers, like for example in a doubly linked list where shared mutable pointers are unavoidable. In those cases, we have to make some reference counting in Rust. This is already implemented in the standard library and therefore not a lot of effort, but it has

a potential impact at runtime. If that worries the programmer, raw pointers can be used instead. In both cases, one has to be careful with the management of the pointers if correct code is required. In general, I would use reference counting and locks to avoid data races and the like, but if code is simple enough that we can proof safety without them, then I would probably use the slightly more efficient method to disable some of Rust's type system by using raw pointers. But if the entire application is done in this style, there is eventually no good reason to use Rust left and I would rather write the whole application in C.
On the other hand, I don't think it will occur that often that the ownership cannot be satisfied with zero costs. But it might take some extra thinking of the programmers every now and then.

4.3 Rust in Barrelfish

Barrelfish and Rust have at least one common goal: They both address the trend of multi-core machines. An interesting question to ask is how well they synergize together. Are they helping each other and make concurrent programming an easy task? Or are they doing both the same job in different ways, therefore making it overly complicated and inefficient?
The answer is, they are solving different instances of the same problem. Barrelfish tackles the problem right at the lowest level of the OS, particularly making the overhead of the kernel in a system of many cores lower.
Rust, on the other hand, helps programmers writing applications. One of the benefits are that race conditions can be completely avoided with as little runtime cost as possible. This is also useful on a multi-kernel system like Barrelfish. When an application wants to use several cores at once, the same problems as with any other OS will occur also in Barrelfish. Thus Rust and Barrelfish do not compete in terms of problems they attempt to solve.
The question becomes more interesting when we consider Rust as a programming language for the kernel itself. The CPU-drivers in Barrelfish run completely single-threaded and do not require any help from a programming language to avoid issues like race conditions. Of course, there are many more features of Rust than those concerning concurrent code, but is the ownership model may be too clunky in a single-threaded environment? So far, nothing like this has been done in Barrelfish, but there has been some research with Rust on embedded systems[18] claiming that there is room for improvement regarding the ease of writing fast Rust code for a single threaded system.

4.4 Micro Benchmark Performance

Since my project's focus is on the comparison of Rust and C, I wanted to evaluate the running time of some functions that are executed repeatedly to handle network traffic. To make sense of the numbers, I looked at the difference between my own implementation, written in Rust, and the existing driver written in C, as mentioned in section 2.2.2. That driver only functions properly in polling mode, thus all my tests are performed with disabled interrupts. Further, the C implementation does not allow for any configurations of hardware features, therefore in the Rust driver all settings are just adjusted accordingly for the tests. In particular, that means that both options for checksum calculations have been turned off and the send ring has a copy located in host memory. Further there only one sender and one receiver ring in used, since none of the two drivers fully supports using multiple rings.
About the Barrelfish version used for the testing, I did some small changes to the C implementation to make it run on the same build of the OS. All tests operate on this version of

Barrelfish, which is based on the internal development repository of Barrelfish at the 21st of February 2017. On top of that, all changes required to run Rust were merged into the OS as well. In particular, all changes that have been done in the initial Rust-Barrelfish port[9] plus the extensions and bugfixes that I made throughout my project. To measure how long the buffer enqueue code takes to be executed in the two versions of the driver, I have looked at the processor internal cycle counter before and after each invocation of the corresponding function handlers provided by the drivers. The difference is the number of cycles spent on the actual driver code plus some small overhead. This is the number that I have looked at for my evaluation and that is used for the data tables below.
I considered three test modes for the micro benchmark testing with different amount of sender activity:

1. **Receiving network traffic only**. No application on the tested machine sends any packets.

2. **One ping per second** is sent by another machine, which will be proceeded and answered by the machine under test. The size of the packets has been **100 bytes**.

3. **Ten pings per second** are sent by another machine, which will be proceeded and answered by the machine under test. The size of the packets has been **1500 bytes.**

The enqueuing of TX buffers and RX buffers have been measured separately. Further, the initial calls to *rx_enqueue* which fill up the buffer descriptor ring are looked at isolated from later invocations of the same function handler.
The data tables below show the Rust implementation is generally on a similar level as the C implementation. The C implementation is clearly faster in test case 1, where no data is sent. The same is true for the receive buffer enqueue calls in test case 2.
For the send buffer descriptor enqueue, the two implementations are pretty much equally fast, but with more traffic the Rust implementation seems to be a bit more efficient.
Interestingly, the Rust driver performed best in test case 3 for both function handlers. Why that is, is a question I could not investigate for long enough to make some well-justified statement about it. But I think it shows that micro benchmarks should be looked at carefully, a small change in the setup may lead to very different results.
From the data presented here, I think the borderline is that Rust can compete with C in terms of pure performance. Or at least the data does not prove the opposite.

Test case 1

	Min	Max	Average	Median	Std deviation
RX C	28	975	74.64	78	29.79
RX Rust	54	2000	176.46	188	64.55

Test case 2

	Min	Max	Average	Median	Std deviation
RX C	28	993	75.40	78	23.68
RX Rust	55	883	159.66	188	59.82
TX C	381	2946	449.79	393	138.72
TX Rust	278	1441	460.27	409	99.3

Test case 3

	Min	Max	Average	Median	Std deviation
RX C	28	970	77.27	82	19.56
RX Rust	56	1235	68.18	56	72.84
TX C	45	2834	448.93	405	84.27
TX Rust	99	1231	334.72	387	166.99

Following on the next pages, there are visualizations of the data. On the x-axis is the numbers of cycles spent for executing the function. On the y-axis is the number of how often a specific cycle count has been measured.
Note that the number of measurement is not exactly the same for the different tests. Instead of putting a hard restriction on the number, I took all data points I logged during a time period. For that reason, the exact numbers on the y-axis should not be compared directly. Never the less, the data plots show nicely how fast the different implementation typically run and how the distribution looks like.

Test case 1: Only receiving data. On the right, the performance to initially set up the receiver ring is shown. This was consistent among all test cases. On the left the performance to enqueue receive buffer descriptors after initialization is presented.

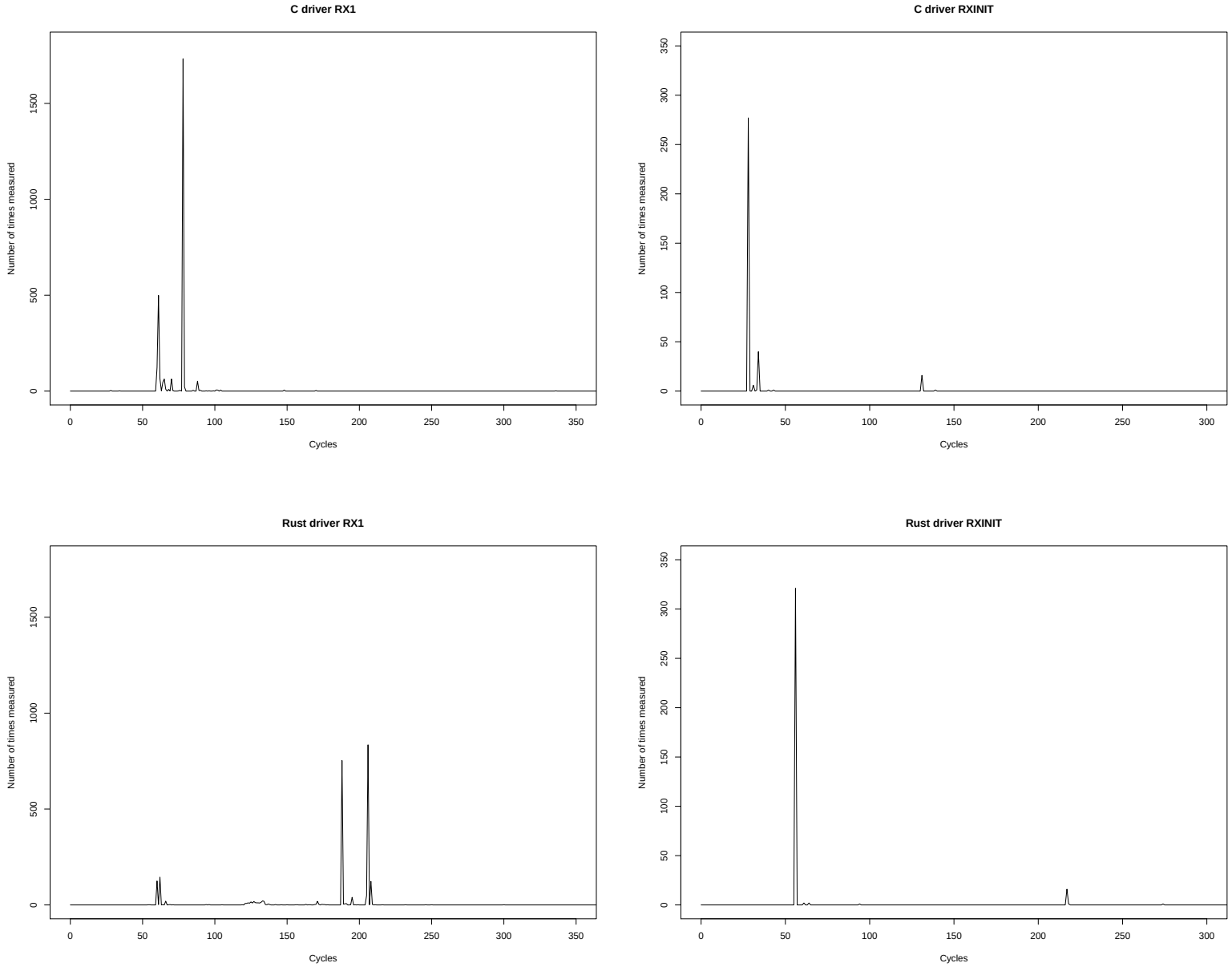

Figure 7: Histograms for micro benchmark test case 1.

Test case 2: Receiving data and handling one ping per second with a size of 100 bytes. On the left, receive buffer descriptors are enqueued, on the right we see the enqueue performance for send buffer descriptors. Both are measured after initialization.

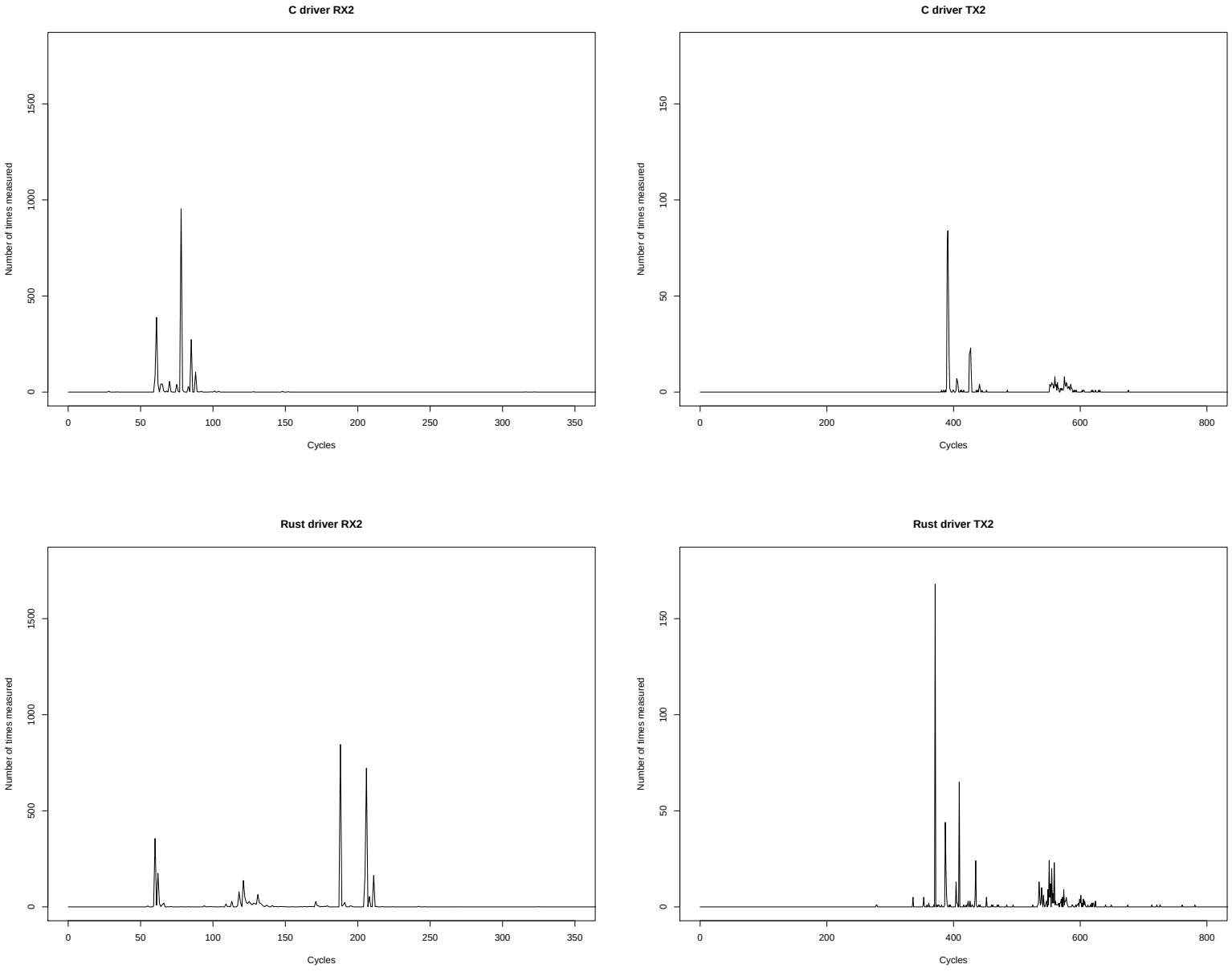

Figure 8: Histograms for micro benchmark test case 2.

Test case 3: Receiving data and handling ten pings per second with a size of 1500 bytes each. On the left, receive buffer descriptors are enqueued, on the right we see the enqueue performance for send buffer descriptors. Both are measured after initialization.

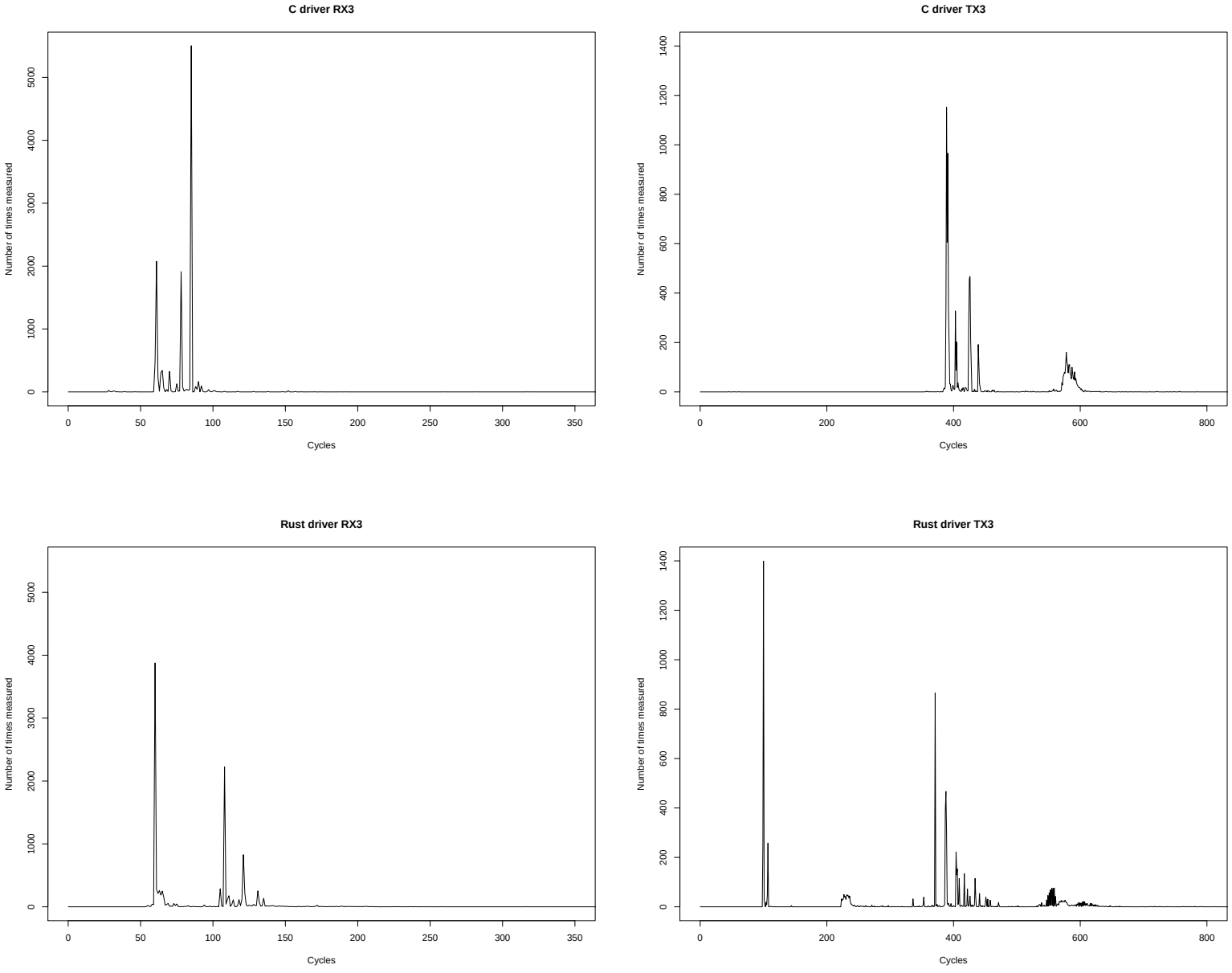

Figure 9: Histograms for micro benchmark test case 3.

Code listings 18 and 19 show the implementation in C and in Rust for the receive buffer enqueue which has been benchmarked. The C code directly operates on the globally available indices. The Rust implementation first checks whether the global state of the driver is existing and then calls a method on the involved ring structure.

```
static errval_t rx_enqueue(uint64_t pa, void* va, void *opaque)
{
    bcm_device_t *bcm = &bcm_dev;
    volatile struct bcm_queue *queue = &bcm->rx_queue;
    if(queue->count == RX_QUEUE_CNT){
        BCM_PRINT_ERROR("No space to add new queue element");
        return EXIT_FAILURE;
    }
    queue->buffers[queue->prodidx] = pa;
    queue->opaque[queue->prodidx] = opaque;
    queue->va[queue->prodidx] = va;
    queue->prodidx = (queue->prodidx + 1) % RX_QUEUE_CNT;
    queue->count++;
    return EXIT_SUCCESS;
}
```

Program code 18: Buffer descriptor enqueue function in C

```
unsafe extern fn rx_enqueue(
        pa: u64,
        va: *const PointerToBuffer,
        opaque: *const Opaque)
-> Errval {
   if let Some(ref mut d) = global_driver {
       d.std_rx.enqueue(pa, opaque, &d.config);
       errval(err_code::SYS_ERR_OK)
   } else {
       errval(err_code::ETHERSRV_ERR_INVALID_STATE)
   }
}
```

Program code 19: Buffer descriptor enqueue function in rust

4.5 Network Benchmarks

4.5.1 Bandwidth and latency

To isolate the capabilities of the card controlled by the implemented driver from as many external effects as possible, I have implemented the required part of the ICMP to answer

ping requests directly in the driver. While all other traffic is still handled by the OS, as usual, ping requests are filtered out and answered by the driver without the OS getting involved. This feature is usually turned off and can be turned on in the same way as hardware features can be configured, that is with a parameter given to the driver at boot time.
For the test in this chapter, generally all configurations are kept in the way they are implemented in the C driver.[30]

Polling mode: First let's have a look at the performance when the driver keeps spinning for incoming network traffic. Generally, this is the easiest mode to run in and the pure network performance should be at its peak.
Running Barrelfish with my driver on one of the rack machines named Brie[31] and five additional Brie machines running Ubuntu 16.04 LTS, I tested the driver under different loads (packets per second) and packet sizes. Regardless of the load, the latency measured was relatively stable with variations in the margin of 10%. With the same load, tests performed directly after each other all resulted in the exact same average and standard deviations for 10'000 pings per test. Repeating the test on different days did only slightly affect the result.
In the following table the measured latency of 10'000 pings sent from a single machine with an interval of 1ms between the pings is presented.

	Min	Max	**Average**	Std deviation
100 bytes payload	0.068ms	0.237ms	**0.07ms**	0.006ms
1400 bytes payload	0.136ms	0.181ms	**0.139ms**	0.012ms

Table 1: Ping latency in polling mode

To push up the load on the card and see how the driver handles the situation, I started 8 processes in each of the five machines running Ubuntu and started the integrated ping flooding command on each of them. For the ping size, I took the maximal size of 1500 bytes that fits into a non-jumbo packet.
Answering all those pings simultaneously, the driver did not show any instability nor have there been any dropped packages.[32]
The processed data for the 40 threads summed up to 983,68 Mbit/s. The theoretical maximum is 1000Mbit's given by the 1000BASE-T Ethernet physical layer. Considering how close to that limit the actual measured values got, I conclude that the Rust implementation of the driver can use the Gigabit card to its full potential when operating in polling mode.

Interrupt handling mode: The usual alternative to polling is that the driver gets notified upon hardware interrupts caused by the NIC. This mode has been implemented for

[30] See section 4.4
[31] See Appendix for the hardware specification
[32] Out of 19'475'171 sent packages over all 40 processes, only 7 were lost. These 7 can be explained because the processes were interrupted with a kill -SIGINT command and there is a high chance that some packages have just been sent when the interrupt occurs.

the driver written in Rust and basic networking functionality with low throughput works stable without any flaws. But with a high number of incoming packets and a resulting high number of interrupts, the system stops responding after some time because the interrupt handler of the driver is not called anymore. As describe in section 2.2.7, the handler should be called by Barrelfish.
Due to time constraints, I could not figure out exactly what is going on, but it looks like the system crashes outside of the driver. At the time of finishing the thesis, this remains an open issue and is discussed as such in section 5.2.
On the positive side, as long as the driver runs stable, the latency is on the same level as we have seen in polling mode. That indicates that the interrupts are forwarded and managed properly, at least with low traffic. For completeness, here is the table with the measured latency. For the smaller packets, the interval between pings was 1ms, for the larger it was 10ms to keep the throughput at a level (1,2Mbit/s) where the system remains stable.

	Min	Max	**Average**	Std deviation
100 bytes payload	0.066ms	0.115ms	**0.069ms**	0.011ms
1400 bytes payload	0.143ms	0.180ms	**0.146ms**	0.008ms

Table 2: Ping latency in interrupt mode

4.5.2 Integration testing

Before I even started the bandwidth tests as described in the previous paragraph, I tested the driver's performance when a typical network centered application runs on the OS. In contrast to the tests on the previous pages, that means the card forwards *all* data to the OS and only sends what it is told by the OS.
The application I used for these test is a web server on Barrelfish. There has already been a module called *Webserver* existing in the Barrelfish source base and I simply used that without any changes. The web server connects to a network file system (NFS) given to it as a parameter, where the data like a *www* folder can be prepared for it by an arbitrary NFS host.
The application running on Barrelfish will then download all files on the server when it starts. Once that is done and then responds to incoming requests on port 80 as any ordinary web server.
All requests to the web server are answered correctly, but the bandwidth and latency have been a bit concerning. Accessing the 20KB index.html file that I used took between 0.5 and 0.8 seconds. A simple download of a 2.5MB file using curl ended up taking up between 45 and 45.5 seconds, which corresponds to a poor bandwidth of 0.44Mbit/s. When increasing the number of threads the throughput is much better but is still far from what one would expect and far from bringing the network card and the driver to its limit. Also, with too many[33] connections up at the same time, the web server gives malformed input to the NetQueueManager causing it to abort. The same happens when I use the C driver.
Additional testing of the networking behavior with some simple pings[34], I noticed that something is wrong. The Round Trip Time (RTT) of ping requests that I measured were about 4ms in some cases, but others took 120ms.
After further investigations described in section 5.1, I concluded that the latency measured

[33] Only up to about 20 connections can be handled correctly.
[34] In that case pings are answered by the Barrelfish network stack, that is lwIP implements ICMP.

in this way has nothing to do with my driver directly, but with the underlying network stack. Because of that, all results that I could get from testing the web server or pings answered by lwIP, are not really relevant for my project. For that reason, I decided to omit any additional details here.

4.5.3 Comparison to C implementation

How much network traffic a driver can handle could be effected by the language choice and one goal of my project was to find out whether the performance is decreased when using Rust instead of C. To answer that question, I wanted to compare the results from this section with the results from the C implementation. But I faced a number of problems that made it hard to make some well-reasoned statement about the performance difference.

1. I could not compare the pure throughput allowed by the two drivers by looking at the performance of user applications. The reason is that the network stack seems to be the bottleneck rather than the driver itself.[35] This seems to be true for both driver implementations.
2. For my driver, I implemented ping request replies in the driver, as discussed in 4.5.1 and I could show what throughput the driver is capable of, but there is no such functionality in the C module. In the associated thesis[10] it is mentioned that for testing purposes the driver has also been extended to answer ICMP directly[36], however I did not have that code.
3. In the thesis about the C module, a measured latency of about 28ms and a calculated bandwidth of 45.6 Mbit/s have been reported. Those values date back to an old Barrelfish version and I could not verify those numbers.

If I just take the numbers presented by the programmer of the C module in his thesis, the highest measured throughput for that driver would be 45.6 Mbit/s, whereas my implementation reached almost the full 1 Gbit/s. But this is obviously not a valid comparison.
To at least get an idea of how the two implementations stand to each other, I performed the same test on the C implementation that I have done with my own implementation. This consists of 10'000 pings of different size and the results are presented in the same form of a table. The result is much better than what was documented with the older Barrelfish distribution.

	Min	Max	**Average**	Std deviation
100 bytes payload	0.086ms	0.214ms	**0.107ms**	0.013ms
1400 bytes payload	0.192ms	0.396ms	**0.214ms**	0.021ms

Table 3: Ping latency of the C implementation (polling)

Notice that for these tests, pings were not answered by the driver, as opposed to the tests done with my implementation. If I turn off the option to answer pings directly in the

[35] This is explained further in section 4.5.2

[36] According to the thesis, the performance has not been improved by answering pings directly in the driver module. This seems a bit odd but I could not verify it without the code.

driver for my implementation, the phenomenon of huge variance[37] in processing time spent inside the network stack occurs as described in section 4.5.2, such that I cannot make any reasonable statement about the latency at all. Since the system using the C driver seems to be unaffected by this, I was at least able to make some decent measurements of the C driver.
The result shows that the latency for pings answered directly by the Rust driver is smaller than the latency for answering pings with lwIP and the C implementation of the driver. But how much of this effect is caused by the difference of answering ping inside the driver or inside lwIP is unclear.
Further, I tested the older driver also for incoming packets of a bandwidth close to 1Gbit/s. Again I used 5 machines running Ubuntu and let 8 processes on each of them send pings to the machine running Barrelfish. Again, no packets have been dropped, but packets have been handled much slower. Within 5 minutes, only 5'714'980 packets were processed. This corresponds to a bandwidth of 228,59 Mbit/s. During this, the latency stabilized at an average of 2.062ms, where the tests with the Rust driver was only at 0.455ms. This is in line with the difference in pure bandwidth.
But to avoid any confusion I must stress again, the Rust driver answers pings on its own, the C driver relies on lwIP. That will be part of the reason why it is slower.

[37] Between 0.7 ms and 240ms

4.6 Conclusion

By building a functional driver for the BCM5704C Gigabit Ethernet card in Rust, I could demonstrate that Rust is a viable choice for programming on a low level and that the language is flexible enough to be adapted to a less mainstream platform like Barrelfish.
Comparing various parts of the implemented driver with an existing implementation written in the more traditional language C, I exemplified what positive and negative impacts the language choice can have.
Being as young as it is, Rust compromises many modern language features with it, while C being as dominant as it has been for a long time offers easier integration into existing environments.
In terms of micro-benchmarks, no tests made as part of this thesis give reason to be concerned about the performance of Rust programs. It seems that C code can, in some cases, perform better than Rust code, but it is also possible the other way around.
The benchmarks focusing on the networking throughput show that the implemented driver is capable of reaching the boundary of the physical layer beneath, which is at 1Gbit/s. Since the speed of a driver beyond that becomes arguably almost irrelevant, this result proofs that Rust code can achieve the required performance for a Gigabit Ethernet driver.
This result gains additional weight when comparing it to the other available Barrelfish driver for the same card, which is written in C and comes to its limit at about a quarter of the full bandwidth because that shows that hitting this limit is not a completely trivial task.
Overall, Rust seems to be a great choice for systems programming for many reasons. Having some safety guarantees is, of course, nice-to-have and would become even more valuable when the product would be used by a large user base.
For me, there have been other benefits which were more valuable. Already in this relatively small project, I could see the advantages of a modularly well-structured code base. When some extension or adaptations to the project are needed, this certainly helps a lot. Rust brings much better tools for that than C, but eventually, it's all up to the programmer.
Not having to think about freeing allocated memory all the time makes programming also much more convenient, compared to C/C++.

5 Unresolved issues

5.1 Latency in network stack

When the driver is working together with Barrelfish's network stack, the latency for network applications is very large and has an immense variance. In section 4.5.2 it is described how this affected the performance for pings and a running web server.
The measured latency seems to follow some patterns. It happened with some consistency, that half of the pings are answered relatively quickly, while the other half are really slow. Sometimes, I observed an alternating behavior for a couple of minutes.
To resolve the issue, I was looking at all possible mistakes inside of the driver first, but nothing that I tried seemed to have an effect on the latency.
Looking at the behavior for different traffic patterns, it seemed to be mostly unaffected by the amount of data flowing through. Sending one small ping per second over a time of 5 minutes resulted in basically the same numbers as sending as many packets of total size 1500 bytes over 1 minute, as visible in the following table.

	Min	Max	Average	**Std deviation**
84 bytes, 1 packet/second	1.038ms	231.034ms	80.769ms	**46.715ms**
1500 bytes, 67 packets/second	0.327ms	239.761ms	83.243ms	**47.627ms**

Table 4: High variance in RTTs for pings

Obviously, the average alone does not really tell much about the data, the standard deviation indicates that and the alternating behavior makes that clear, too. But looking at all measured RTTs after sorting shows a nearly uniform distribution when collecting data for ten minutes, despite the alternating behavior. All data points collected are visualized in the following plots, after they have been sorted by the RTT.

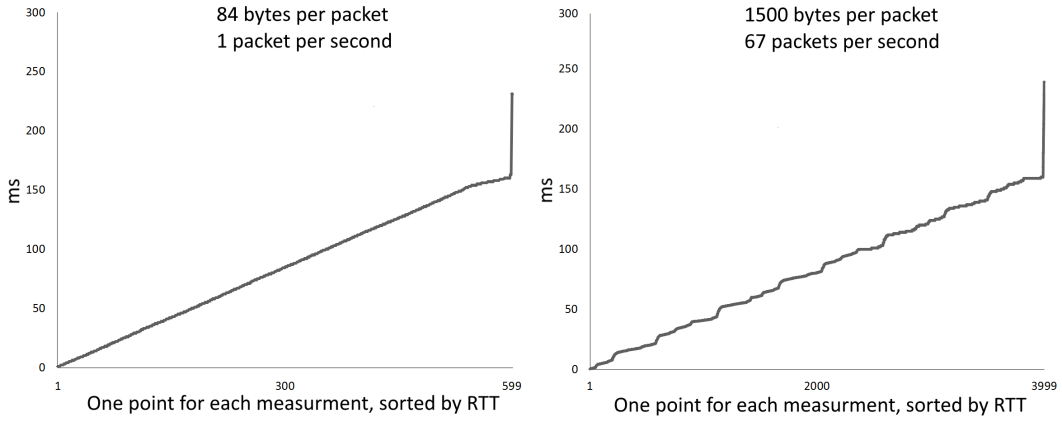

Figure 10: Sorted RTTs measured in the experiment

Since I couldn't explain this behavior being caused by my driver, I measured the time between the moment when the NetQueueManager receives a buffer and the moment when

the according answer is given back to the driver. It turned out that the variance in the RTT was entirely caused by the time between those two events. The same alternating behavior was observable there and in some cases, the delay for generating the answer to a ping request in the network stack was only slightly shorter than the total RTT.
I have not been able to resolve this issue. To measure the maximum throughput of the driver I have implemented a mechanism that circumvents the problem by answering pings directly in the driver. This mechanism does not affect how non-ping traffic is being handled by the system and it can be turned on and off.

5.2 Dealing with a high number of interrupts

The driver is able to operate in an interrupt processing mode, in which it only checks for received data after the registered interrupt handler has been called by Barrelfish.
For most of the development time, the driver was only able to operate in polling mode because it took me some time to integrate the driver into Barrelfish properly so that the driver will receive all interrupts that are addressed to it. Only shortly before the end of the project, the interrupt handler has been called upon arriving data consistently.
The open issue is that the system becomes unstable as soon as the network traffic load goes above a couple of Mbit/s. At some point, the network stack stops functioning. In particular, the interrupt handler of the driver is not being called anymore, thus no incoming data will be processed after that. It also seems to be the case that outgoing data does not reach the driver anymore.
I could not investigate it for long enough to track down the problem to its roots, but here are the facts that I could gather while debugging and testing with console output and pings:

- The higher the traffic data per second, the faster the systems stops functioning. The critical amount where it starts to break down within few seconds is around 15-20 Mbit/s.
- The behavior appears to be non-deterministic. Sometimes the system keeps responding for more than 5 minutes with the same load that it was breaking down in a matter of seconds the test run before.
- No crash inside the driver happens and the driver software keeps running even when the machine is unreachable for other network devices.
- The system does not recover by itself after it stopped forwarding any interrupts.
- Several minutes after networking stopped working, Barrelfish reports that the netd is down.

My guess is that some part of Barrelfish's network stack stops working with a high load of interrupts, a next step would be to have a closer look at the netd and also how interrupts are routed exactly.
But it could also have other reasons, maybe the driver acknowledges interrupts in a faulty way. Or it is also possible that the card stops generating interrupts for some reason, a wrong hardware configuration could cause that.
Unfortunately, I was not able to analyze that further before the deadline of my project.

6 Related Work

6.1 The System: Barrelfish and its Network Stack

6.1.1 Barrelfish BCM5704 Gigabit Ethernet Driver

As part of Philipp Gamper's bachelor thesis, a Barrelfish driver for the Broadcom BCM5704 Gigabit Ethernet card has been built. While good performance can be achieved in polling mode, interrupt processing is still problematic. At the time that project has been finished, spring 2015, there have also been some problems with the underlying network stack on Barrelfish and how interrupts are handled. Therefore it is likely that the problems came also from the underlying system, rather than solely from the driver implementation itself.
This work serves as a direct performance comparison between a driver written in C and one in Rust. Because both drivers can manage the same hardware, the results are well suited for such a comparison. The detailed evaluation can be found in chapter 4,

6.1.2 Arrakis

Arrakis[21] is an OS based on Barrelfish. It aims to show how to reduce the OS overhead for applications found in data centers with high I/O data throughput. The approach is to improve performance by removing the kernel from the I/O data path as much as possible, moving more work to the application and the hardware and thereby avoiding the extra copy of the data inside of the kernel. They also show how Arrakis can still perform the usual tasks of an OS to provide security and isolation properties.
Applications written to the POSIX interface can be run on Arrakis without further adaptations. By using Arrakis instead of Linux, they obtain a significant performance boost in terms of latency and data throughput, although the kernel will still have to copy the data. When modifying the application specifically for Arrakis, the OS can be removed from the data path entirely, resulting in even further increased performance.

6.1.3 Dragonet

Most modern NICs come with many hardware features performing tasks traditionally done by software, to reduce the workload on the CPU. The control over these features is almost exclusively in the device drivers. An implication of this is that the control logic has to be rewritten for each and every device, even though it looks very similar in many cases. Another problem is that the OS has no control over those features when it is all handled in the driver. This concept - resources that are managed without the control of the OS - seems to be against the principle concept of what abstractions an OS should provide.
The network stack called Dragonet tries to address these problems and an implementation of it has been developed for Barrelfish. [25][16] The new approach of Dragonet is to model the network card in a physical resource graph (PRG) and the required packet processing in a logical protocol graph (LPG), both of which are data flow graphs. To describe these graphs, a DSL called Unicorn[24] has been developed.
Once the graphs are built, they are ready for the OS to be optimized. Theoretically, it can compute which hardware features should be enabled for any given state of the network stack for the maximal performance.
With this approach, the OS gains control over the hardware features, and further, decisions

on how to use them are based on the knowledge of the whole system.
Comparing the achieved latency and throughput of the implementation of Dragonet on Barrelfish against a modern Linux network stack showed that the former is certainly able to compete, and often even better than the latter.

6.2 Other OS and programming language research

6.2.1 Singularity: Rethinking the Software Stack

In the first decade of the current century, Singularity[14] was in active development at Microsoft Research. It is an OS that was created to enable research in that particular area. Performance was only second priority and application and driver compatibility has been abandoned completely. That choice allowed for a liberated exploration of new programming languages and verification tools.
The three architectural features characterizing Singularity are:

- Software isolated processes (SIP): Instead of traditional processes, user applications run in SIPs which do not share data with each other. Each SIP is associated with a set of threads, a number of memory pages and with a security identity. This security identity maps to OS security attributes and provides information hiding and failure isolation.
- Contract based channels between the SIPs: All communication between SIPs has to go through channels. Static contracts define what kind of messages are allowed in different states of the channel, which is useful for static analysis and software verification.
- Manifest-based programs: A program on Singularity comes with a static manifest which defines required resources, desired capabilities and dependencies on other programs. An application cannot be started in other ways than by invoking its application manifest.

6.3 Rust: A New Systems Programming Language

6.3.1 Cyclone

Cyclone[15] is a C dialect that has been created about two decades ago with the goal to prevent typical bugs like buffer overflows with static language guarantees and some inserted runtime checks. At the same time, the low-level programming style of C has been preserved. Today, Cyclone is no longer developed. It is still available online[38], but it does not run on 64-bit machines without further effort.
In many ways, the ideas pursued by Cyclone are similar to what Rust does today. Indeed the development of Rust has been influenced by Cyclone.

6.3.2 Servo

Today, all well-known browsers[39] are based on an engine written in C++. Complete control of the underlying hardware features is important, therefore many other languages are not a

[38] `https://cyclone.thelanguage.org/`
[39] Google Chrome, Mozilla Firefox, Microsoft Internet Explorer, Microsoft Edge, Apple Safari, Opera, ...

suitable choice.
A new browser engine, called Servo [40], is currently developed in Rust. The goal is to have at least the same performance as already existing browser engines, but being more adapted to these days multiple core machines. This should lead to better performance on mobile phones and even lower their energy consumption.
From an experience report by the Servo developer[4], we can learn about the consequences of using the new systems language for this project. The safety properties of Rust help in many ways, for instance, they make use-after-free bugs impossible. Because of that, the developers of Servo think it is justified to spend a little bit more time to write code that passes Rust's type checker.
One problem for developing Servos is that Rust can't compile against C++ code. On the other hand, it is possible to compile against C code. Therefore they could work around this problem by writing some wrappers in C, but that sometimes leads to performance loss because cross-language inlining is not working, yet. However, Servo's developers already stated that they will be able to use cross-language inlining between C++ and Rust and they intend to use it in the future.
Another point the Servos developers noted in their experience report concerns unsafe code. They think it would be generally helpful to the language if there were some additional annotations which make is possible to prove basic properties about unsafe code.

6.3.3 Porting Rust to Barrelfish

In spring 2016, the programming language Rust has been ported to Barrelfish as the MSc. thesis of C. Foellmi.[9] The general goal of the project was to make Rust a viable choice for development in Barrelfish. For that it was necessary to provide an implementation for all OS specific parts of the standard library and the build tools used in Barrelfish needed adaptation as well.
Thanks to the successful completion of that project, students like myself are now able to explore Rust as a systems programming language within the research context currently available at ETHZ. Another research project, Timley Dataflow[41], also used this port to run a Rust version on Barrelfish.
As far as the conclusion of that project goes, the main question answered was how well the promises made by the language can be held in practice. The type system offers the possibility for safer and more readable systems code. Its zero-cost abstractions make sure there is no drawback in performance. All of that should desirably be possible without any major burdens in the development process.
C. Foellmi shows in which aspects those promises are not quite accomplished. He mentions several issues with Rust as a systems language. Especially integrating C code over an FFI leads to several difficulties which could be handled better by the language. Furthermore, some zero-cost abstractions seem to have some hidden costs when looking at the binaries more closely. Besides, there are some smaller issues, often because they could not be addressed by the developers, yet. For instance the namespace of Rusts' macros has been criticized in the thesis.
Looking at it about one year later, I can see that Rust experiences good progress and that the maturity of the language is increasing steadily. Never the less, there are still many open issues and the macro namespace problem, in particular, is one of them. [3]

[40] https://servo.org/
[41] https://github.com/frankmcsherry/timely-dataflow

6.3.4 Rust2Viper: Building a Static Verifier for Rust

Viper[20] is a verification infrastructure developed at ETHZ. An intermediate language called Silver is used to formulate permissions. Using this intermediate representation, two different verification back-ends are available for static analysis of the programs. One is named Silicon and uses symbolic execution, the other is called Carbon and uses verification condition generation.
In early 2016, Viper has been extended with a Rust front-end.[11] Besides translating Rust programs to Silver programs, it also generates some predicates and permissions automatically. To do so, it is taking advantage of Rust's strong type system.
The Rust front-end is yet missing traits, type parameters, and closures. But basic verification is possible and with some extra effort in client code, Viper is capable of verifying properties like the validity of memory regions pointed to by raw pointers.

6.3.5 Shared mutable state in Rust

With the ownership model, Rust gives strong memory safety guarantees at runtime. However, programmers have to provide enough information to the compiler, so it can prove those safety properties for every case.
Embedded OS developers claimed that for a single-threaded environment the current type system of Rust is not expressive enough.[18] They show examples where they had written some code they knew was safe on one thread, but because mutable aliasing is strictly forbidden in Rust, it wouldn't compile. They suggest adding the notion of *Execution Contexts* to Rust, giving programmers the possibility to express that two procedures will always run on the same thread. The goal of this would be to enable shared mutable states within one thread safely.
The paper was published in 2015, but so far, nothing similar to the suggested *Execution Contexts* has been added to Rust and I could not find any Request For Comments (RFCs) suggesting to do so. I think many programmers are not really interested in this feature because there is already a way in Rust to have safely shared mutable state. Rust's standard library offers structures for reference counting and cells for internal mutability. The combination of the two[42] can be used to gain the shared, safe and mutable state. But this solution is not perfect, it relies on runtime checks and is not given as compile-time guarantee, which the concept of *Execution Contexts* tries to deliver.

6.3.6 GPU Programming in Rust

Programming a GPU can be challenging, but for some applications, the results will vastly outperform the CPU. After LLVM gained a new compilation target, Nvidia's low-level virtual instruction set PTX, it opened the door for advanced language features in GPU kernels. Since the Rust compiler, rustc, uses LLVM as its backend, this made enabling low-level GPU programming in Rust much easier.
Upon this, a team of computer scientist extended Rust with support for GPU kernels.[12] In their publication, they show how high-level abstractions from Rust allow programmers to write GPU kernels in a less traditional and more modern way. A strength described is that both system-level abstractions, as well as more convenient high level code, can be written in the same language and therefore projects get more homogeneous.
The performance evaluation showed that the performance of GPU kernels created by pure

[42] The declared type in Rust would be: Rc<RefCell<T>>

Rust can compete with hand-written OpenCL kernels, but of course, the latter is generally a bit more efficient.

6.3.7 Reenix: Implementing a Unix-Like Operating System in Rust

The project Reenix[19] was a, mostly successful, attempt to implement the greater part of an operating system using Rust. Reenix builds on top of the teaching OS Weenix[27], which allowed the developer to keep some of the lowest level code of the kernel, such as memory allocators. Due to time limitations, the project has not been completed. But while implementing some key features of an OS, including a process model, a virtual file system, and basic ATA hard disk and TTY drivers, many insights about the strengths and weaknesses of Rust as a systems programming language have been gained.
As one would probably expect, the higher abstractions provided by Rust, compared to C, made the development generally easier. The powerful macro system is also described as very useful. Furthermore, custom compiler plugins have also been used for the implementation of Reenix.
For the development of a kernel, the standard library of Rust has also been pointed out to be much stronger than what is available in C, because the latter is mostly an interface to kernel features assumed to be existing already.
The powerful static type, lifetime and borrow checker is also described as useful because it helps to eliminate entire classes of errors. There was nothing mentioned that the development process of Reenix would have been slowed down particularly by those checkers.
Beside smaller critique on the usability of Rust and some suggestions on how to make a few cases easier to code, there was a major problem pointed out in the report of the project: In Rust, memory allocation is assumed to be infallible. In other words, when the system runs out of memory and a memory allocation fails, the application will abort. In user space, that scenario might be rare enough to overlook that problem. In kernel space, this is not acceptable at all. In the kernel, we absolutely need a way to handle the case when the system runs out of memory.[43] The solution to change the allocator function of Rust to have a return value signifying whether it was successful seems to be quite straightforward at first, but that would need adaptations in the entire standard library. Of course, that would also affect programmers who work completely in user space. For them, there would not be a real benefit, but a lot of additional code would have to be written and rewritten to handle the out-of-memory case whenever a standard collection is initialized or possibly extended. Therefore, this might not be the best solution for the problem. Yet another reason against this approach is the tremendous amount of work to adapt the standard library accordingly.
With the acceptance of #RFC1398 [2], it will be possible in Rust to write and use custom allocators in the future. In fact, it has been possible in the nightly build already, even before the RFC was accepted. A custom allocator can handle out of memory failures better. Instead of aborting the program, it can panic, which can then be recovered by the caller. However. the #RFC1398 does not define explicit changes to the behavior of the standard library. The standard containers still use the default allocator. Unless parameters are implemented for them to change the allocator, it will be difficult to take advantage of them in a kernel environment.
For the project Reenix, an ugly workaround has been used which also decreases the performance of memory allocation.

[43] At least that is true in the traditional model of operating systems. In Barrelfish, there is no dynamic memory allocation done inside the kernel.

When comparing the overall performance of a C implementation of Weenix with Reenix in some very basic speed test, the Rust implementation was slower by a factor between two and three, which is, of course, a major drawback. But this result should be taken carefully, as the implementations do not do the exact same thing, and there are mentioned workarounds in the Rust implementation.

7 Future Work

7.1 Dedicating hardware queues to user applications

To fully leverage the features of Barrelfish and its network stack, a driver should be completely split in two. One part initializing and maintaining the shared resources of the card, the other part detached for each queue. In that way, each queue would be mapped to a hardware queue and the data could be filtered for the corresponding applications directly on hardware. The second part is also called *queue driver*.
The driver created as a part of this bachelor thesis brings all functionalities to run with one to sixteen queues. But because the design fot the driver was heavily oriented on the existing driver, I have not implemented queue drivers separately. Instead, the one and only instance of the driver loops through all queues and manages all of them. It could still be useful for prioritizing some traffic over other, but in most cases, the preferred configuration will be to have one single queue.
Extracting all relevant code into a queue driver module would be the next step to improve the functionalities of the driver and make it synergize better with Barrelfish.

7.2 Runtime hardware feature optimization

As with most modern network cards, the BCM5704C has a number of optional hardware features which may effect its runtime behavior in different ways. Whether these effects are desirable is dependent on many factors unknown at compile-time. For example, if the card performs TCP/UDP and IP checksum offloading, it can reduce the load on the CPU. But that might not be supported by the OS or maybe it is simply not helpful because the CPU is idle most of the time.
There are attempts to optimize the use of all available hardware features in a NIC at runtime[16], but the topic is still subject to research[25] and has not made its way into the industry quite yet.
In the new driver, there is a configuration struct that holds static information like the configured Maximum Transmission Unit (MTU) for the card, but also runtime configurable values indicating which hardware features should be active. The configuration object is stored in the state of the driver instance and therefore available in all methods of the main driver module and references to it can be borrowed from submodules if their behavior depends on the configuration.
Currently, the driver takes the following arguments that have an effect on its behavior:

- Polling mode vs. Interrupt mode
- Answer ICMP ping request directly in the driver
- Keep a copy of the send buffer descriptor ring(s) in host memory
- TCP/UDP checksum offloading
- IP checksum offloading

So far, the settings are set at initialization via the argument list provided to the driver. However, the settings for checksum offloading have corresponding setter functions implemented. The driver will respect changes through them and can dynamically change between different modes of operation.

The other settings are determined once and for all when the driver is initialized. Because the fields are private and no setter function is provided, programmers have currently no way of changing them at runtime. But given the design with the configuration struct, adding more dynamic behavior would require minimal work, since implementing a setter function in the *Config* module would be enough to enable dynamic changes. With that, all relevant code that depends on the runtime configuration already has access to it.
Even extending the driver to allow dynamic mode switch between polling and interrupt handling should be quite simple considering the current design of the driver.
So as possible future research project, if someone wants to experiment how hardware features can be turned on and off at runtime by an OS, the Rust implementation of the BCM5704C driver brings the basics to allow such analysis with a small amount of work on the driver side.

7.3 Static analysis and formal methods

The seL4[17] demonstrates impressively that an entire micro-kernel can be proved correct formally, without giving up on performance. That raises the question for me if we could not do the same with an entire network stack implementation, including a driver like the one created in this bachelor project. At least to me, having an end-to-end connection where all involved parts have been formally proved seems rather compelling.
Rust comes with a number of built in safety guarantees which hold for code which is not marked as unsafe. The problem is that one could do arbitrarily bad things in unsafe code and therefore the system as a whole does not bring any safety properties for free. But the strong type system of Rust might still be useful to apply formal methods.
Recent efforts in program verification for Rust[11] make it possible to use top edge static analysis tools on Rust code. This is especially interesting for unsafe code, since it could cover the holes in safety introduced by the unsafe code sections.
For a future research project, it seems interesting to investigate how verification of a system like a NIC is affected by using Rust as programming language. Potentially, proving all unsafe sections to be correct in combination with Rust's type system would implicitly also prove properties like memory safety for all sections. Further I could imagine that following a Rust idiomatic design works well along with static verifying tools. It should be helpful to come up with the right permissions and predicated when the code itself features modularity, information hiding and tools like iterators.

8 Lessons learned

Throughout the project, I have encountered a large number of topics that were new to me and I certainly improved my knowledge in various areas.
The center of my project has been Rust, but in that context, I have not only mastered the language itself, but I have also learned more about the role of a programming language in the development process. Here I think I have only just begun my journey of programming language exploration. On one side, I solidified my personal penchant towards Rust, but I also gained a strong motivation to learn more about other languages' features. In particular, I am looking forward to learning about the newest features of C++.
Besides the new language that I could explore, I was also lucky to work on a very interesting OS and I got some insight into the development process of the same. This changed my view towards OSs in general, and I also learned many technical details about OSs.
Staying on the technical side, I also improved my understanding of networking substantially. Although I knew all involved protocols before, understanding them on a theoretical level is still very different from actually implementing them and getting to see all pitfalls arising when working with the involved devices.
Last but not least, I learned that translating code directly is not a very sensible idea. Because I had a working implementation of a driver written in C, my naive first approach was to just make a one-to-one translation to Rust code. But of course I did not fully understand the entire code of the driver and I have to admit that I did not read the programmer's guide thoroughly enough when I started coding. The result was, that I have done a lot of extra work to translate some code patterns from C to Rust when it later turned out that this pattern was not necessary at all. Or in one case, I decided to change the design structure completely to meet my ideas of how the abstractions should be done, resulting in the deletions of several hundred lines of code.
Concluding, I think I should have started my implementation by reading the programmer's guide, following it step by step. The existing driver could have served me very well to resolve problems as they appear, especially when it comes to OS specific aspects. But that should have come into play much later in the development process, rather than right at the start as I did it now.

9 Appendix

9.1 Figures

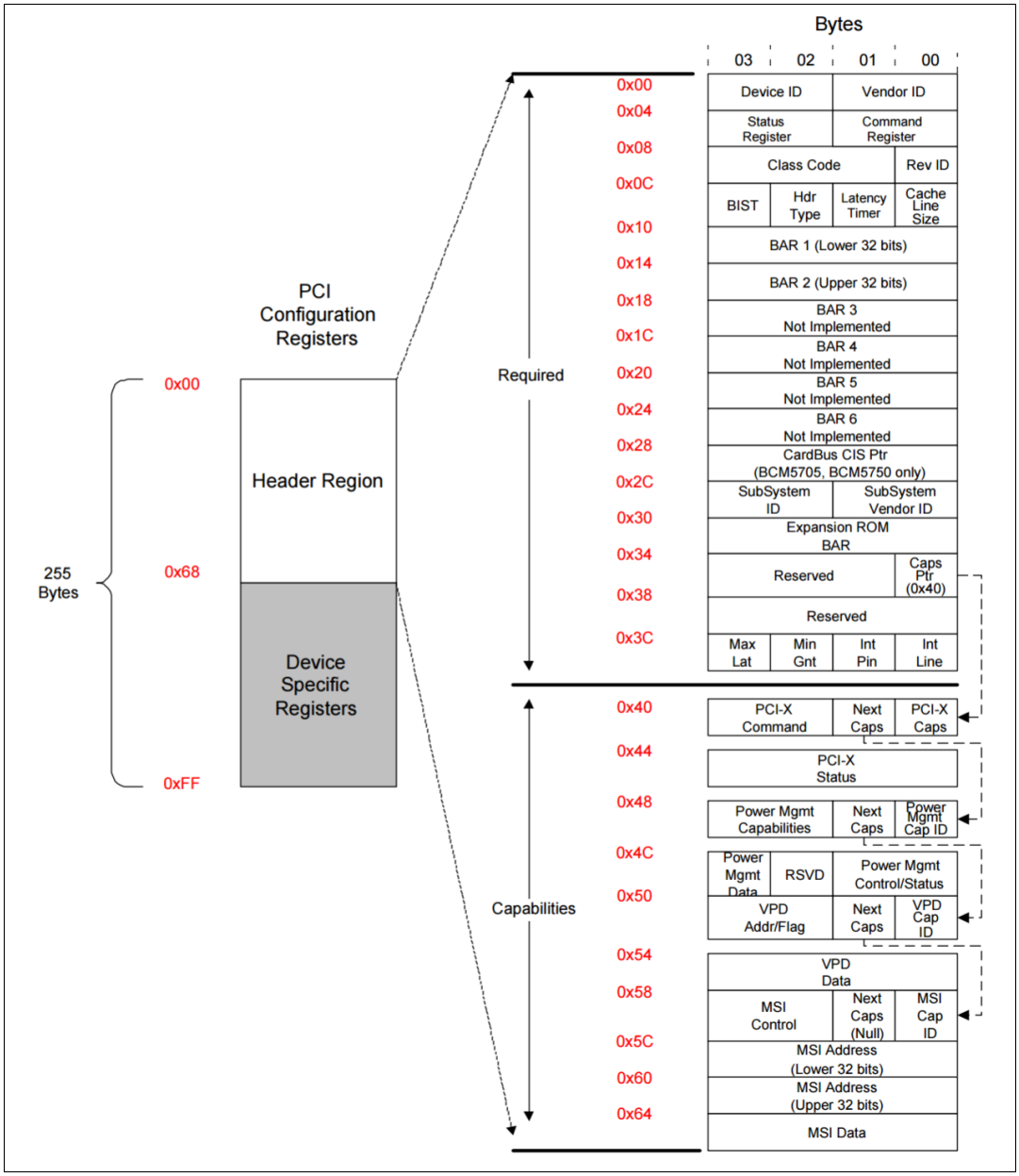

Figure 11: PCI header region from figure 65 in the programmer's guide

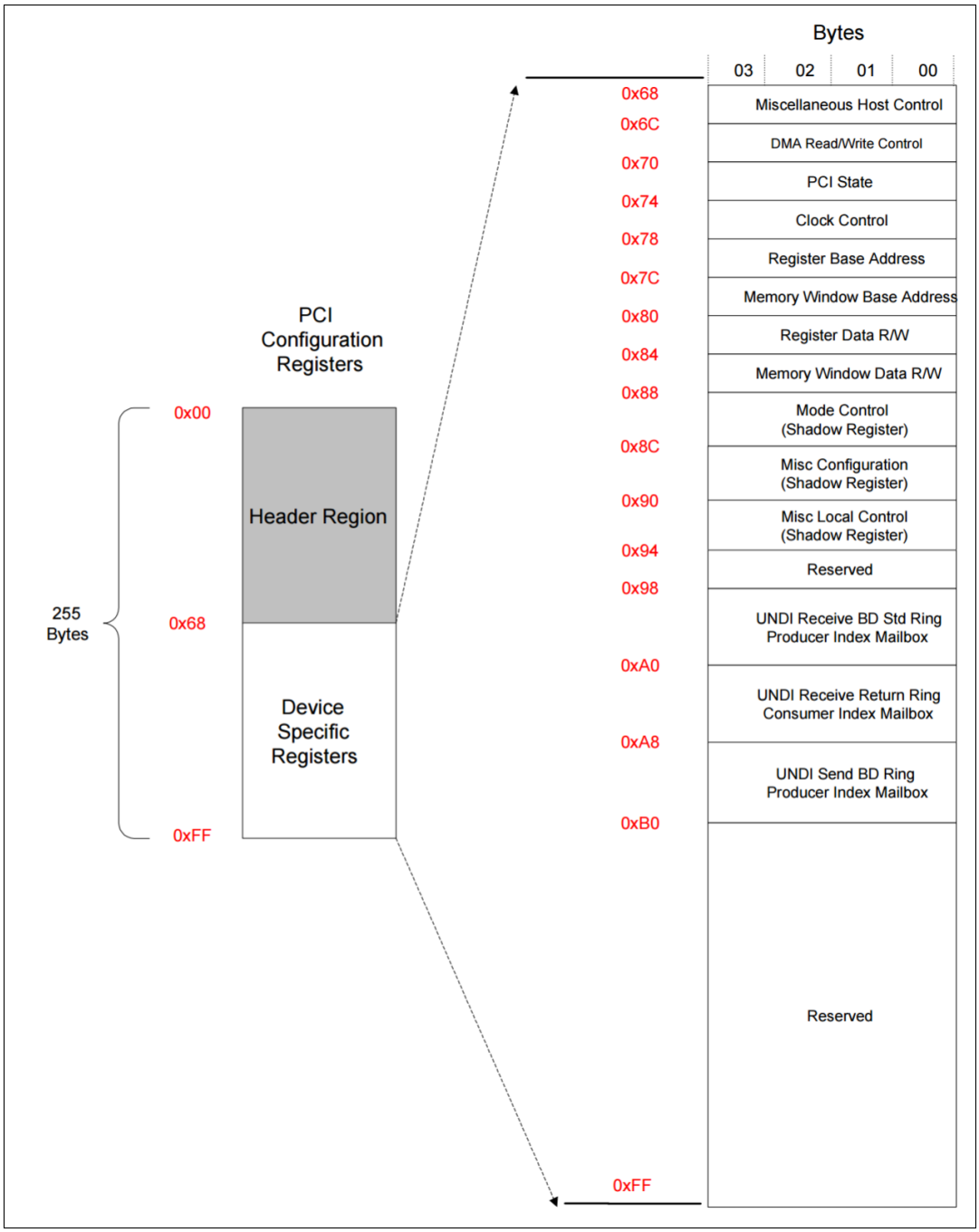

Figure 12: PCI device specific region from figure 66 in the programmer's guide

9.2 Code

9.2.1 DMA Array

```rust
pub struct DmaArray<T> {
    va: *mut T,
    pa:  u64,
    length: usize,
}
impl<T> DmaArray<T> {
    /// allocates memory for n instances of T
    /// memory is zeroed (but not initialized with valid data)
    pub fn new(n: usize) -> Result<DmaArray<T>, ErrorCode>{
        let size =  ::std::mem::size_of::<T>() * n;
        match dma_frame_calloc(size) {
           Ok((va, pa)) => Ok(
                DmaArray{
                    va: va as *mut T,
                    pa: pa,
                    length: n,
                }
            ),
            Err(e) => return Err(e),
        }
    }
    /// Removes an element stored in the array allocated with new_array()
        ///  and takes ownership, the internal data is overwritten with zero
        ///  and leaves an uninitialized slot behind.
    /// Checks index boundary at runtime and panics if it is violated.
    pub fn remove(&mut self, i: usize) -> T {
        assert!(i < self.length);
        unsafe{
            // Here I need to return the element stored inside and remove it from
            //  the managed memory. To achieve that in a safe way, I copy the
            //  element to a new location on the stack, zero out the managed memory
            //  and lastly move the stack allocated element as return value
            let size = ::std::mem::size_of::<T>();
            let mut result : T = ::std::mem::uninitialized();
            ::std::ptr::copy_nonoverlapping(self.va.offset(i as isize), &mut result, 1);
            *self.va.offset(i as isize) = ::std::mem::zeroed();
            result
        }
    }
    #[inline]
    pub fn len(&self) -> usize {
       self.length
    }
```

```
    #[inline]
    pub fn get_physical_address(&self) -> u64 {
       self.pa
    }
}
impl<T> Index<usize> for DmaArray<T> {
    type Output = T;
    /// Array access to the stored elements allocated earlier.
    /// Checks index boundary at runtime and panics if it is violated.
    fn index(&self, i: usize) -> &T {
        assert!(i < self.length);
        unsafe{
            &*self.va.offset(i as isize)
        }
    }
}
impl<T> IndexMut<usize> for DmaArray<T> {
    /// Array access to the stored elements allocated with new_array().
    /// Checks index boundary at runtime and panics if it is violated.
    fn index_mut(&mut self, i: usize) -> &mut T {
        assert!(i < self.length);
        unsafe{
            &mut *self.va.offset(i as isize)
        }
    }
}
/// Debug trait for printing the contained values
impl<T : ::std::fmt::Debug> ::std::fmt::Debug for DmaArray<T> {
    fn fmt(&self, f: &mut ::std::fmt::Formatter) -> ::std::fmt::Result {
        try!(write!(f,
                        "DMA Array at va 0x{:x} = pa 0x{:x} contains the entries: \n",
            self.va as u64,
            self.pa,
        ));
        for i in 0..self.len() {
            try!(write!(f, "{:?};   ", self[i]));
        }
        Ok(())
    }
}
```

Program code 20: The DmaArray struct with Debug and Index traits.

9.2.2 RCB copying

Two of the three RCBs have to be copied into the memory window. Writes are only possible through 32-bit register writes, but the RCB also has fields which are 16-bits and 64-bits wide.

```c
#define RCB_OFF_ADDR_HI     offsetof(struct bcm_rcb, host_addr)
#define RCB_OFF_ADDR_LO     offsetof(struct bcm_rcb, host_addr) \
                                                + sizeof(uint32_t)
#define RCB_OFF_FLAGS       offsetof(struct bcm_rcb, flags)
#define RCB_OFF_MAX_LEN     offsetof(struct bcm_rcb, max_len)
#define RCB_OFF_NIC_ADDR    offsetof(struct bcm_rcb, nic_ring_addr)
/* Configure Send Ring RCB0 */
rcb = &bcm->tx.rcb;
bcm5704_memwin_write_32(dev, SRAM_SEND_RCB + RCB_OFF_ADDR_HI,
                BCM_ADDR_HI(bcm->tx.frame));
bcm5704_memwin_write_32(dev, SRAM_SEND_RCB + RCB_OFF_ADDR_LO,
                BCM_ADDR_LO(bcm->tx.frame));
bcm5704_memwin_write_32(dev, SRAM_SEND_RCB + RCB_OFF_FLAGS,
                rcb->flags | rcb->max_len << 16);
bcm5704_memwin_write_32(dev, SRAM_SEND_RCB + RCB_OFF_NIC_ADDR,
                rcb->nic_ring_addr);
```

Program code 21: Writing the ring control block to the card internal memory region in C

```rust
/* Inside the impl{} block of the RCB struct */
pub fn write_to_misc_mem(&self, dev: &Device, address: u32) {
  for i in 0..4 { // struct is 4 x 32 bits large
    super::read_write::write_mem_std(
      dev,
      address + i as u32 * 4, // raw pointer arithmetic for 32bit steps
      unsafe {
        // pointer offset for 32bit steps (transmute needs unsafe)
        *::std::mem::transmute::<&BcmRcb, *const u32>(self).offset(i)
      }
    );
  }
}
/* Calling above function from the buffer descriptor ring module */
self.rcb.write_to_misc_mem(dev, SRAM_SEND_RCB);
```

Program code 22: Writing the ring control block to the card internal memory region in Rust

9.3 Hardware specification

The Systems Group at ETHZ conventionally names their rack machines after cheese brands. The type of machines named Brie contain a Broadcom BCM5704C Gigabit Ethernet card each and those are the machines I used for all tests described in this thesis.
The detailed specification for Brie machines is:

CPU	Dual Core AMD Opteron Rev E 90nm
Motherboard	MSI-9245
System memory	8 × 512MB
Chipset	AMD 8111 I/O bridge
	AMD 8131 PCI-X Hub

10 List of graphics and code samples

List of code blocks

List of Figures

11 Acronyms

ARP Address Resolution Protocol

ATA AT Attachment

BAR Base Adress Register

CPU central processing unit

DHCP Dynamic Host Configuration Protocol

DMA direct memory access

DNS Domain Name Service

DSL domain specific language

ETHZ Eidgenoessische Technische Hochschule Zuerich

FFI foreign function interface

GDB GNU Debugger

GNU GNU's not Unix

GPU graphics processing unit

ICMP Internet Control Message Protocol

IoT Internet of Things

IP Internet Protocol

LAN Local Area Network

LLVM Low Level Virtual Machine

LPG logical protocol graph

lwIP lightweight IP

MAC Media Access Control

MI Management Interface

MSI Message Signaled Interrupt

MTU Maximum Transmission Unit

netd networking daemon

NFS network file system

NIC network interface controller

NQM NetQueueManager

OOP	object oriented programming
OS	operating system
OSI	Open Systems Interconnection
PCI	Peripheral Component Interconnect
PHY	Physical Layer
PnP BIOS	Plug and Play BIOS
POSIX	Portable Operating System Interface
PRG	physical resource graph
PTX	Parallel Thread Execution - an intermediary assembler language by NVIDIA
RCB	Ring Control Block
RFC	Request For Comment
RISC	Reduced Instruction Set Computer
RTT	Round Trip Time
SKB	system knowledge base
TCP	Transmission Control Protocol
TTY	talk to you / teletypewriter - refering to a text based terminal
UDP	User Datagram Protocol
UNDI	Universal Network Device Interface
VLAN	Virtual Local Area Network
WOL	Wake on LAN

12 References

[1] Remove documentation references to garbage collection (issue #13987), May 2014. https://github.com/rust-lang/rust/issues/13987. 10

[2] Allocator api (rfc #1398), April 2016. https://github.com/rust-lang/rfcs/blob/master/text/1398-kinds-of-allocators.md. 6.3.7

[3] Tracking issue for 'macro naming and modularisation' (rfc #1561), August 2016. https://github.com/rust-lang/rust/issues/35896. 6.3.3

[4] B. Anderson, L. Bergstrom, D. Herman, J. Matthews, K. McAllister, J. Moftt, S. Sapin, and M. Goregaokar. Experience report: Developing the servo web browser engine using rust. Technical report, Mozilla Research and Indian Institute of Technology Bombay, May 2015. 6.3.2

[5] A. Baumann, P. Barham, R. Isaacs, T. Harris, S. Peter, T. Roscoe, , A. Schpbach, and A. Singhania. The multikernel: A new os architecture for scalable multicore systems. In *22nd Symposium on Operating Systems Principles*, pages 29–44. Association for Computing Machinery, Inc., October 2009. 2.1

[6] Broadcom Corporation, 5300 California Avenue; P.O. Box 57013; Irvine, CA 92617. *Programmers Guide BCM57XX: Host Programmer Interface Specification for the NetXtreme Family of Highly Integrated Media Access Controllers*, January 2008. 2.2, 2.2.1

[7] T. R. Community. *The Rust Standard Library*. The Rust Team, February 2017. 4.2.1

[8] A. Dunkels. Design and implementation of the lwip tcp/ip stack, 2001. 2.1.5

[9] C. Foellmi. Os development in rust, November 2015. 2.1.6, 3.1, 3.2, 3.3.1, 4.4, 6.3.3

[10] P. Gamper. Barrelfish bcm5704 gigabit ethernet driver, May 2015. 2.2.2, 2

[11] F. Hahn. Rust2viper: Building a static verifier for rust, April 2016. 6.3.4, 7.3

[12] E. Holk, M. Pathirage, A. Chauhan, A. Lumsdaine, and N. D. Matsakis. Gpu programming in rust: Implementing high level abstractions in a systems level language. In *27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, pages 315–324, 2013. 6.3.6

[13] Q. Hui and L. Qi. Implementation of lwip tcp/ip protocol stack based on s1c33e07. In Y. Wu, editor, *Software Engineering and Knowledge Engineering: Theory and Practice. Advances in Intelligent and Soft Computing, vol 114.*, pages 635–642. Springer, Berlin, Heidelberg, 2012. 2.1.5

[14] G. Hunt and J. Larus. Singularity: Rethinking the software stack. *ACM SIGOPS Operating Systems Review*, 41:37–49, April 2007. 6.2.1

[15] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, , and Y. Wang. Cyclone: A safe dialect of c. In *USENIX Annual Technical Conference*, pages 275–288. USENIX Association Berkeley, CA, October 2002. 6.3.1

[16] A. Kaufmann. Efficiently executing the dragonet network stack, 2014. 6.1.3, 7.2

[17] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4: formal verification of an os kernel. In *22nd Symposium on Operating Systems Principles*, pages 207–220. Association for Computing Machinery, Inc., October 2009. 2.1.2, 7.3

[18] A. Levy, M. P. Andersen, B. Campbell, D. Culler, P. Dutta, B. Ghena, P. Levis, and P. Pannuto. Ownership is theft: Experiences building an embedded os in rust. In *8th Workshop on Programming Languages and Operating Systems*, pages 21–26. Association for Computing Machinery, Inc., October 2015. 4.3, 6.3.5

[19] A. Light. Reenix: Implementing a unix-like operating system in rust. Technical report, Brown University, Department of Computer Science, April 2015. 6.3.7

[20] P. Mueller, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. In *VMCAI 2016 Proceedings of the 17th International Conference on Verification, Model Checking, and Abstract Interpretation - Volume 9583*, pages 41–62. Springer-Verlag New York, Inc. New York, NY, January 2016. 6.3.4

[21] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: the operating system is the control plane. In *11th USENIX Symposium on Operating Systems Design and Implementation*, 2014. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/peter. 6.1.2

[22] T. Roscoe. *Barrelfish Technical Note 003: Hake*. Systems Group, Department of Computer Science, ETH Zurich, CAB F.79, Universitatstrasse 6, Zurich 8092, Switzerland, April 2010. 2.1.3

[23] T. Roscoe. *Barrelfish Technical Note 002: Mackerel*. Systems Group, Department of Computer Science, ETH Zurich, CAB F.79, Universitatstrasse 6, Zurich 8092, Switzerland, March 2013. 2.1.6

[24] P. Shinde, A. Kaufmann, K. Kourtis, and T. Roscoe. Modeling nics with unicorn. In *7th Workshop on Programming Languages and Operating Systems*, November 2013. 6.1.3

[25] P. Shinde, A. Kaufmann, T. Roscoe, and S. Kaestle. We need to talk about nics. In *14th Workshop on Hot Topics in Operating Systems*, 2013. 6.1.3, 7.2

[26] A. Singhania, I. Kuz, and M. Nevill. *Capability Management in Barrelfish*. Systems Group, Department of Computer Science, ETH Zurich, CAB F.79, Universitatstrasse 6, Zurich 8092, Switzerland, Dezember 2013. 2.1.2

[27] O. S. T. A. Sta. Weenix, March 2015. http://cs.brown.edu/courses/cs167/docs/weenix.pdf. 6.3.7

[28] H. Sutter. Trip report: Summer iso c++ standards meeting (oulu). `https://herbsutter.com/2016/06/30/trip-report-summer-iso-c-standards-meeting-oulu/`, June 2016. Accessed: 2017-03-09. 1

[29] G. Zellweger. *Device Drivers in Barrelfish.* Systems Group, Department of Computer Science, ETH Zurich, CAB F.79, Universitatstrasse 6, Zurich 8092, Switzerland, Dezember 2013. 2.1.4

literature

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Eigenständigkeitserklärung

Die unterzeichnete Eigenständigkeitserklärung ist Bestandteil jeder während des Studiums verfassten Semester-, Bachelor- und Master-Arbeit oder anderen Abschlussarbeit (auch der jeweils elektronischen Version).

Die Dozentinnen und Dozenten können auch für andere bei ihnen verfasste schriftliche Arbeiten eine Eigenständigkeitserklärung verlangen.

Ich bestätige, die vorliegende Arbeit selbständig und in eigenen Worten verfasst zu haben. Davon ausgenommen sind sprachliche und inhaltliche Korrekturvorschläge durch die Betreuer und Betreuerinnen der Arbeit.

Titel der Arbeit (in Druckschrift):

Barrelfish NIC driver in Rust

Verfasst von (in Druckschrift):
Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich.

Name(n):	**Vorname(n):**
Meier	Jakob

Ich bestätige mit meiner Unterschrift:

- Ich habe keine im Merkblatt „Zitier-Knigge" beschriebene Form des Plagiats begangen.
- Ich habe alle Methoden, Daten und Arbeitsabläufe wahrheitsgetreu dokumentiert.
- Ich habe keine Daten manipuliert.
- Ich habe alle Personen erwähnt, welche die Arbeit wesentlich unterstützt haben.

Ich nehme zur Kenntnis, dass die Arbeit mit elektronischen Hilfsmitteln auf Plagiate überprüft werden kann.

Ort, Datum	**Unterschrift(en)**
Zürich, 19. März 2017	J. Meier

Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich. Durch die Unterschriften bürgen sie gemeinsam für den gesamten Inhalt dieser schriftlichen Arbeit.